2. Algorithmus

2.1 Einleitung

Algorithmen sind wichtig für Informatiker und Computerprogrammierer - aber ihre Bedeutung geht erheblich darüber hinaus: Der Begriff des **Algorithmus** stellt eine der **wichtigsten zentralen Ideen der Mathematik** dar; in seiner Bedeutung nur vergleichbar etwa mit dem Begriff der **Funktion**. **Algorithmen** sind nicht zuletzt für die Allgemeinbildung von Bedeutung.

Die **Algorithmen** stehen in enger Wechselwirkung mit weiteren zentralen Themen unseres geistigen Lebens, insbesondere mit der **Philosophie**, **Mathematik**, **Informatik**, **Heuristik, Kulturgeschichte** und der allgemeinen **Bildung**.

Die Entstehung von **Algorithmen** hat zunächst einmal gar nichts mit Computern zu tun. Die **Algorithmen** sind sehr viel älter als die Computer. Die **Algorithmisierung geistiger Prozesse** ist jedoch die Voraussetzung dafür, daß dies Operationen von Computern übernommen werden können.

Eines der ersten dokumentierten Beispiele für einen mathematischen Standard-Algorithmus ist die auf einer Keilschrift festgehaltene babylonisch-sumerische Methode des ***Wurzelziehens*** (**Verfahren von Heron**, im **60-er System** in Keilschrift in den Stein geschlagen, **ca. -1700**)

vgl. J.Ziegenbalg; Algorithmen von Hammurapi bis Gödel; Harri Deutsch, 2.Aufl., 2007

Weitere frühe schriftliche Belege für Algorithmen bietet **Euklids** Buch ***"Die Elemente"***, die ab ca. **-400** verfasste Gesamtdarstellung des mathematischen Wissens seiner Zeit (die neben vielen anderen natürlich auch den **"Euklidischen" Algorithmus** enthält).

Euklids Werk war eines der erfolgreichsten Lehrbücher in der Bildungsgeschichte der Menschheit; es wurde bis ins vorige Jahrhundert hinein als Lehrbuch für Mathematik verwendet - ein Lehrbuch mit einer "Lebenszeit" von über 2000 Jahren.

Der Begriff des **Algorithmus** geht zurück auf die Angabe ("al Khowarizmì") des in Zentralasien - im heutigen Uzbekistan - liegenden Geburtsorts des persisch-arabischen Gelehrten

Abu Ja'far Mohammed ibn Musa al-Khowarizmi,

(etwa 783-850), der im neunten Jahrhundert (ca. 825) ein wissenschaftsgeschichtlich äußerst einflussreiches Lehrbuch zur Verbreitung der **"indischen Ziffern"**, also der Zahldarstellung im **Zehnersystem**, geschrieben hat.

Die Art und Weise, wie wir heute die Zahlen schreiben und mit ihnen umgehen, wurde entscheidend von diesem Buch beeinflusst, das mithin neben **Euklids "Elementen"** einen der wichtigsten Ecksteine in der Bildungsgeschichte der Menschheit darstellt.

Allgemein kann gesagt werden, daß Mathematik in ihrer Entstehungsgeschichte **Algorithmik** war.

Abu Ja'far Mohammed ibn Musa al-Khowarizmi (nach [Zemanek 1981] und [Wußing 1989])

ins Deutsche übertragen: Mohammed, Vater des Ja’far, Sohn des Mose, geboren in Khowarizm

Bedingt durch die Schwierigkeiten bei der Sprachübertragung wird der Name auch geschrieben als *al-Charismi*, *al-Chorezmi*, *al-Chwarazmi*, *al-Hwarizmi*, *al-Khorezmi*, *al-Khuwarizmi*, *al-Khwarizmi*

2.2 Aspekte des algorithmischen Arbeitens

Die Algorithmik besitzt zwei Aspekte:

- **Entwurf**, **Konstruktion** von (neuen) Algorithmen
  (Damit ist im weiteren Sinne auch das **Verstehen**, **Nachvollziehen**, **Analysieren** und **Verbessern** vorgegebener, insbesondere **klassischer Algorithmen** gemeint.)

Soll die Beschäftigung mit der **Algorithmik** aber nicht nur theoretischer Natur bleiben, so gehört zum anderen aber auch die

- **Abarbeitung** von fertigen, vorgegebenen Algorithmen zum Gesamtgebiet der Algorithmik.

Der **Entwurf von Algorithmen** ist eine **höchst kreative Aktivität**; ihre Abarbeitung dagegen meist eine sehr langweilige Sache. Deshalb gehörte zur Algorithmik schon immer der Versuch, Maschinen zur Abarbeitung von Algorithmen zu konstruieren.

Die **Universalmaschine** zur Abarbeitung von Algorithmen ist heute der **Computer**

Algorithmisches Arbeiten beinhaltet eine **tiefe** und **zeitaufwendige Analyse** des zu algorithmierenden Sachverhaltes.

Eine **operative** und **vollständige Vorgehensweise** entsprechend der Grundfrage
" ... was passiert, wenn .... "
ist charakteristisch für die Algorithmierung.

Heuristische Entwürfe sind wesentliche Bestandteile der Algorithmierung.

Auch das **systematische Variieren** von **Parametern** und **Einflussfaktoren** und das **Experimentieren** gehören zu einer operativen Vorgehensweise bei der Algorithmierung.

Algorithmierung heißt, einen Gegenstand **eigenständig** und **konstruktiv** zu erschließen.
Jeder Algorithmus ist eine "**Konstruktionsvorschrift**" zur Lösung eines bestimmten Problems.

Algorithmisch vorzugehen heißt, sich die Lösung eines Problems schrittweise aus **Elementarbausteinen** aufzubauen.

Algorithmen werden **experimentell** und **beispielgebunden** getestet.

Algorithmische Lösungen vermeiden oft technisch und begrifflich aufwendige Methoden der Formel-orientierten Mathematik zugunsten wesentlich **elementarerer iterativer** bzw. **rekursiver** Vorgehensweisen.

Die Methode der **Simulation** beruht auf der Modellierung realer Sachverhalte mittels Algorithmen und deren Abarbeitung auf Computern zur Erlangung neuer Erkenntnisse aufgrund von **Experimenten**.

Algorithmisches Arbeiten kann sehr **unterschiedliche Wissensfelder** miteinander verbinden.

2.3 Begriff des Algorithmus

Definition 1: Ein ***Algorithmus*** ist eine **eindeutige Beschreibung** eines **endlichen Verfahrens** zur **Lösung** einer bestimmten **Klasse** von Problemen.

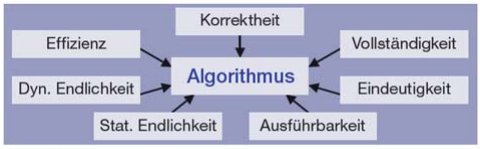
Definition 2: Ein ***Algorithmus*** ist eine **Berechnungsregel**, die aus mehreren **elementaren Schritten** besteht, die in einer bestimmten **Reihenfolge** ausgeführt werden müssen. Die Anwendung dieser Berechnungsregel führt nach einer **endlichen** Anzahl von Schritten zu einem **Funktionswert** oder einer **Datenmenge**.

Einfaches Beispiel für einen Algorithmus:

Suchen der größten Zahl einer Menge von Zahlen

```
Schritt 1: lies erste Zahl

Schritt 2: initialisiere z mit der gelesenen Zahl

Schritt 3: lies die nächste Zahl

Schritt 4: wenn diese Zahl größer z, dann setze z auf diese Zahl

Schritt 5: wenn noch Zahlen vorhanden, dann gehe nach Schritt 3

Schritt 6: gib z aus
```

Allgemein können einem Algorithmus folgende Merkmale zugeordnet werden:



Eigenschaften eines Algorithmus

a.) Ein Algorithmus muß von einer **Maschine** durchgefürt werden können. Die für den Ablauf des Algorithmus benötigte **Information** muß zu Beginn vorhanden sein.

b.) Ein Algorithmus muß **allgemeingültig** sein. Die Größe der Datenmenge, auf die der Algorithmus angewandt wird, darf nicht eingeschränkt sein.

c.) Der Algorithmus besteht aus einer **endlichen** Reihe von **Einzelschritten** und Anweisungen über die **Reihenfolge** (= **Statische Endlichkeit**). Jeder Schritt muß in seiner **Wirkung eindeutig** und **genau definiert** sein.

d.) Ein Algorithmus muß nach einer **endlichen Zeit** (und nach einer endlichen Anzahl von Schritten = **dynamische Endlichkeit**) enden. Für das Ende des Algorithmus wird eine **Abbruchbedingung** formuliert.

e.) Unter **Korrektheit** versteht man, daß der Algorithmus die seiner Entwicklung zugrunde liegende **Spezifikation** erfüllt, d.h. daß er genau das und nur das Ergebnis korrekt liefert, welches in der Aufgabenstellung spezifiziert wurde.

f.) **Vollständigkeit** heißt, daß neben allen erforderlichen Schritten und Aktionen auch **alle relevanten Vor- und Rahmenbedingungen**, die eine erfolgreiche Ausführung des Algorithmus erst ermöglichen, behandelt werden.

g.) Unter **Effizienz** versteht man die Eigenschaft eines Algorithmus, seinen Zweck unter bestmöglicher Ausnutzung aller benötigter Ressourcen zu erfüllen (Speicherplatz, Laufzeit, ...)

Definition 3: Ein ***Algorithmus*** ist eine **endliche**, präzise **Vorschrift**, die von einem konkreten System (Prozessor) in ***Elementarschritten*** ausgeführt werden kann. Jeder ***Elementarschritt*** muss dabei vom System stets in ***endlicher Zeit*** durchgeführt werden können.

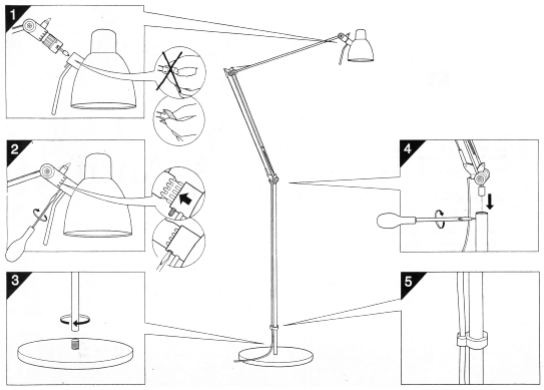
Ein ***Terminierender Algorithmus*** bricht stets nach **endlich** vielen Elementarschritten ab.

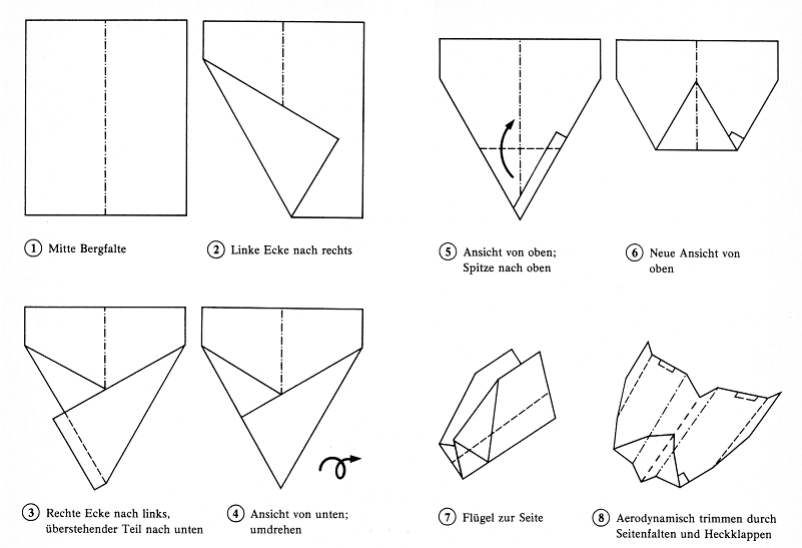
Alltagsalgorithmen

Die Alltagswelt, die uns umgibt, ist voller Algorithmen. Wenn Sie z. B. vom Automaten einen *Latte Macchiato* trinken wollen, dann führen Sie eine Folge koordinierter Abläufe aus, die schließlich das gewünschte Produkt erscheinen lassen oder nicht. Die Lösungsvorschrift hierzu haben Sie sich durch Lernen und üben erworben und in Ihrem Gehirn abgespeichert.

Sie tragen also den Algorithmus "Kaffee am Automaten auswählen" zusammen mit vielen anderen Algorithmen in Ihrem Kopf herum.

Viele Alltagsalgorithmen sind schriftlich formuliert, z.B. Bedinungsanleitungen, die Berechnung der neuen Kfz-Steuer oder Kochrezepte, aber auch Wegbeschreibungen in einem Wanderführer oder Gesetze. Andere Algorithmen sind in Form eines Bildes formuliert, z.B. die *Aufbauanleitung für eine Lampe* oder die *Anleitung zum Falten eines Papierfliegers*:





Algorithmus "Gulaschsuppe zubereiten"

Zu manipulierende Objekte:
350 g Rindfleisch, 3 Zwiebeln, 50 g Schweineschmalz, 15 g Mehl, 1 kleine Dose Tomatenmark, 3/4 l Wasser, Salz, Paprika, Majoran.

Hilfsobjekte:
Herd, Kochgeschirr

Anweisungen:
Fleisch und Zwiebeln würfeln
in Schmalz andünsten
mit Mehl bestäuben und kurz anrösten
Tomatenmark zugeben
mit Wasser auffüllen
garen
mit Salz, Paprika und Majoran abschmecken.

An diesem Beispiel wird deutlich, daß ein Algorithmus aus zwei Teilen besteht:

- einem **Deklarationsteil**, in dem die zu manipulierenden Objekte deklariert werden
- einem **Aktionsteil**, in dem die auszuführenden Aktionen in Form von Anweisungen beschrieben werden.

Algorithmen spielen in der (schriftlich tradierten) Kultur- und Wissenschaftsgeschichte der Menschheit als gesetzliche Vorschriften und Handlungsprotokolle aller Art eine wichtige Rolle.

Ein frühes Beispiel hierfür ist der ***Codex Hammurapi*** aus der Zeit um **-1780** (Hammurapi war 6. König von Babylonien (Mesopotamien), Gesetz 2 - sinngemäß):

"Wenn jemand einen Mann beschuldigt und der Angeklagte zum Fluss geht und in den Fluss springt, und wenn er dann untergeht, so soll der Ankläger sein Haus in Besitz nehmen. Aber wenn der Fluss beweist, dass der Angeklagte unschuldig ist und wenn dieser unverletzt entkommt, dann werde der Ankläger zum Tode verurteilt, während derjenige, der in den Fluss gesprungen ist, vom Hause des Anklägers Besitz ergreifen soll."

2.4 Algorithmieren, Programmieren

Unter **Algorithmierung** versteht man das Konstruieren bzw. Finden eines **terminierenden Algorithmus *a***, der eine gegebene Funktion berechnet.

Unter einem **Programm *p*** versteht man die **Niederschrift eines Algorithmus *a(p)*** in einer (so genannten) **Programmiersprache**. Ein Programm ist somit ein **Wort über einem Alphabet** und ihm kann die jeweilige **Funktion** $f_{a(p)}$ als **Bedeutung** zugeordnet werden.

Eine **Programmiersprache *P*** besteht aus einer Menge ***S*** von (so genannten) **Programmen**, die eine **Bedeutung** besitzen.

Unter **Syntax** einer Programmiersprache versteht man die **Definition der Menge** der zugehörigen **Programme**. Zur formalen Definition werden häufig **Grammatiken** und **Syntaxdiagramme** genutzt.

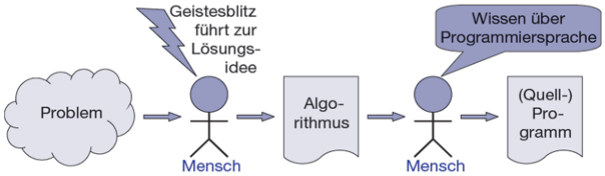
Unter **Semantik** einer Programmiersprache versteht man die ***Abbildung***, die den Programmen die **Funktion** zuordnet, die dem beschriebenen Algorithmus entspricht:

Semantik : Menge der Programme der Programmiersprache → Menge der Funktionen

Diese Abbildung wird zumeist verbal beschrieben oder mit Hilfe eines **Compilers** präzisiert.

Unter **Programmierung** versteht man heute in der Regel das
- **Algorithmieren** und
- das **Formulieren** des **Algorithmus** in einer **Programmiersprache**.

Gelegentlich wird auch unter Programmierung nur das Formulieren eines gegebenen Algorithmus in einer Programmiersprache verstanden.



Von der Problemstellung über den Algorithmus zum (Quell-)Programm

Algorithmierungs-/Programmier-Aufgabe:
Ein **Algorithmus** beschreibt, wie aus **Eingabedaten** die zugehörigen **Ergebnisse**, d.h. **Ausgabedaten**, ermittelt werden können. Das bedeutet, dass damit eine **Abbildung** (oder Funktion) von den Eingabedaten in die Ausgabedaten realisiert wird:

Gegeben: **Funktion** $f: A \rightarrow B$
Gesucht: **Terminierender Algorithmus**, der die Funktion f **berechnet**.
(Evtl. wird noch die Formulierung in einer Programmiersprache gefordert)

In der ***Berechenbarkeitstheorie*** nennt man eine Funktion f **berechenbar**, wenn es einen *Algorithmus* A gibt, der die Funktion berechnet, d.h. $\forall x(f(x) = f_A(x))$

Die **Funktion**, die ein Algorithmus **berechnet**, ist gegeben durch die ***Ausgabe***, mit der der Algorithmus auf eine ***Eingabe*** reagiert.

Der **Definitionsbereich** der Funktion ist die **Menge der Eingaben**, für die der Algorithmus überhaupt eine **Ausgabe** produziert. Wenn der *Algorithmus* nicht terminiert, dann liegt die *Eingabe* nicht im Definitionsbereich.
Falls Algorithmus für **beliebige Zeichenreihen** terminiert, dann heißt Funktion **entscheidbar**.

Man sagt, der Algorithmus A *berechnet* die Funktion $\mathbf{f}: T \rightarrow \mathrm{N}$ mit $T \subseteq N^k$, wenn A bei Eingabe von $(\mathbf{n_1, ..., n_k}) \in T$ nach einer **endlichen** Zahl von Schritten den Wert $\mathbf{f(n_1, ..., n_k)}$ ausgibt und damit **terminiert**, und bei Eingabe von $(\mathbf{n_1, ..., n_k}) \in N^k \setminus T$ **nicht terminiert**.

Eine Funktion **f** heißt ***berechenbar***, wenn es einen Algorithmus A gibt, der sie berechnet.

Terminiert ein Algorithmus A, dann sagt man auch, dass der Algorithmus die Funktion **f** ***berechnet.***

Die Begriffe **Algorithmus** und **Berechenbarkeit** werden synonym verwendet.

Beispiel: Berechnung der Tankkosten in Abhängigkeit vom Preis und dem Tankvolumen:

f (preis, volumen) = preis * volumen, falls preis > 0.0 [€/l] und volumen >= 0.0 [l] ,
sonst nicht terminierend

$\boldsymbol{T}$ = { (preis, volumen) ∈ R² , preis > 0.0 **und** volumen >= 0.0 }, z.B. (1.20, 40) ∈ $\boldsymbol{T}$

$\boldsymbol{N^2} \setminus \boldsymbol{T}$ = { (preis, volumen) ∈ R² , preis <= 0.0 **oder** volumen < 0.0 }, z.B (1.20, -10) ∉ $\boldsymbol{T}$

d.h. f (1.20, 40) = 1.20 * 40 = 60.00 [€]
f (1.20, -10) = nicht terminierend

Ein möglicher **Algorithmus A** für obige Funktion **f** (in C / C++ - Syntax):

```
double preis, volumen;                      // Deklarationsteil
cin >> preis;                               // Tastatureingabe von preis
cin >> volumen;                             // Tastatureingabe von volumen
do {
     if( preis > 0.0  &&  volumen >= 0.0 )  // wenn preis und volumen die Bedingung erfüllen,
          break;                            // dann verlasse do - while - Schleife
} while( preis <= 0.0 || volumen < 0.0 );   // solange preis oder volumen die komplementäre
                                            // Bedingung erfüllen, laufe durch Schleife
cout << preis * volumen;                    // Tastaturausgabe
```

Bemerkungen: Für die praktische Nutzung wäre im obigen Algorithmus die Verlagerung der Tastatureingabe in das Innere des Schleifenkörpers sinnvoll.

Das angegebene C++ - Programm ist noch nicht vollständig und bzgl. der Ein- und Ausgabe nicht "robust".

Beispiel für algorithmisch nicht lösbare Probleme:

Beispiel 1: Nicht zu jeder Funktion **f:** $\boldsymbol{N} \rightarrow \boldsymbol{N}$ ($\boldsymbol{N}$ sei die Menge der natürlichen Zahlen) gibt es einen Algorithmus, mit dessen Hilfe bei vorgegebenem Argument der Funktionswert ermittelt werden kann.

Begründung: Die Gesamtheit aller Algorithmen ist ***abzählbar unendlich,*** die Menge der obigen Funktionen ist jedoch ***überabzählbar unendlich***.

Bemerkungen:
Eine Menge wird *abzählbar unendlich* genannt, wenn es eine umkehrbar eindeutige Abbildung zwischen dieser Menge und den *natürlichen Zahlen* gibt.

Die Menge der reellen Zahlen ist *überabzählbar unendlich*. Das bedeutet, daß es **keine** *umkehrbar eindeutige Abbildung* gibt, die jede reelle Zahl auf eine natürliche Zahl abbildet.
(weiteres bei Georg Cantor, Begründer der Mengenlehre)

Beispiel 2: Das **zehnte Hilbertsche Problem**: Gesucht ist ein Algorithmus, mit dessen Hilfe man für jede *diophantische Gleichung* entscheiden kann, ob sie lösbar ist oder nicht.

Eine *diophantische Gleichung* ist eine Gleichung der Form $\mathbf{p(x_1, x_2, ...,x_n) = 0}$, wobei **p** ein **Polynom** in den Variablen $\mathbf{x_1, x_2, ...,x_n}$ mit **ganzzahligen** Koeffizienten ist.

Bei einem "diophantischen" Problem sind **ganzzahlige Lösungen** gesucht, welche bei Einsetzung an die Stelle der Variablen $\mathbf{x_1, x_2, ...,x_n}$, das Polynom zu "Null" machen.

Diese Einschränkung der Lösungsmenge ergibt einen Sinn, wenn Teilbarkeitsfragen beantwortet werden sollen oder wenn bei Problemen in der Praxis nur ganzzahlige Lösungen sinnvoll sind, z. B. Stückzahlverteilung bei der Herstellung von mehreren Produkten.

Beispiele für diophantische Gleichungen:

$X^2 - Y = 0$ besitzt als **Lösung** die Zahlenpaare (1,1), (2,4), (-2,4), (3,9), (-3,9), ... allgemein: $(\pm n, n^2)$.

$X^4 + Y^2 + Z^{20} = -7$ besitzt ***keine*** **Lösung**, da die linke Seite der Gleichung immer größer oder gleich Null ist.

$3X = 4$ besitzt ***keine*** **Lösung**, da bei diophantischen Gleichungen nur **ganzzahlige** Lösungen gesucht sind.

Diophantische Gleichungen sind nach dem griechischen Mathematiker *Diophant von Alexandrien/Diophantos*, um 250, benannt .

1970 bewies Juri Wladimirowitsch Matijassewitsch, dass die Lösbarkeit einer *diophantischen Gleichung* **unentscheidbar** ist.

Eine spezielle *diophantische Gleichung* ist Gegenstand des **"Großen Fermatschen Satzes"** (1637): Die Gleichung $x^n + y^n = z^n$ besitzt für ganzzahlige x, y, z ungleich 0 und natürliche Zahlen $n > 2$ **keine** Lösungen.

Dieser Satz konnte erst 1994 von den britischen Mathematikern Andrew Wiles und Richard Taylor bewiesen werden. Vgl. Simon Singh; Fermats letzter Satz; DTV-Verlag

Ein Programm ***P*** heißt genau dann **korrekt bzgl. einer Funktion**, wenn der vom Programm beschriebene Algorithmus ***A(P)*** genau diese Funktion berechnet, d.h. wenn gilt

$$\forall x(f(x) = f_{A(P)}(x)$$

Es gibt weitere Präzisiierungenen des Begriffes **Algorithmus**.

Kurt Gödel (*rekursive Funktionen*), **Alan Turing** (*Turing Maschine*), **Alonzo Church** (*Lambda-Kalkül*) entwickelten unterschiedliche Fassungen des Begriffes "**Berechenbarkeit"**.

Die **Churchsche These** (jede Funktion, die in irgendeiner Art und Weise berechenbar ist, kann auch durch eine Turing-Maschine berechnet werden) besagt, daß alle bisher bekannten formalen Fassungen des Begriffs der **Berechenbarkeit** gleichwertig sind. Diese These ist kein innermathematischer Satz und kann auch **nicht** mit mathematischen Mitteln bewiesen werden.

Jeder Algorithmus wird mit dem Programm einer "***Turing-Maschine (TM)***" identifiziert.
Bemerkung: Die ***Turingmaschine*** ist ein ***abstrakter Rechner***, der aus einem unendlich langen Speicherband mit unendlich vielen sequentiell angeordneten Feldern besteht. In jedem dieser Felder ist zu jedem Zeitpunkt genau ein Zeichen gespeichert, auch Blanksymbole sind erlaubt. In einer einzigen Bewegung kann die ***TM*** das Zeichen auf dem einen vom Schreib-Lese-Kopf bearbeiteten Quadrat lesen und, basierend auf diesem Zeichen und ihrem gegenwärtigen Zustand, dieses Zeichen durch ein anderes ersetzen, ihren Zustand wechseln und sich um eine Position nach links oder rechts bewegen. Jeder reale Computer kann genau das berechnen, was eine ***TM*** berechnen kann. Die ***TM*** wird deshalb als **abstraktes Standardmodell eines Computers** angesehen.

Definition der **Entscheidbarkeit**: Falls eine **Aussage A** in **endlich** vielen Schritten **herleitbar** oder **widerlegbar** ist, dann heißt A ***entscheidbar***, ansonsten ***unentscheidbar***.

Kurt Gödel (1906 - 1978) wies 1931 die **Unentscheidbarkeit der Arithmetik** nach, auch als ***1. Gödelschen Unvollständigkeitssatz*** bekannt:

Es gibt **keinen Algorithmus**, mit dessen Hilfe man für jede arithmetische Aussage (inerhalb der PEANO-Axiome) in endlich vielen Schritten entscheiden kann, ob sie wahr oder falsch ist.

D.h. es gibt keinen Algorithmus, mit dessen Hilfe man genau die gültigen arithmetischen Aussagen herleiten kann, d.h. innerhalb eines formalen Systems sind nicht alle Aussagen beweisbar.

2. Gödelscher Satz: Kein formales Axiomensystem, welches mindestens die additiven und multiplikativen Eigenschaften der natürlichen Zahlen beinhaltet, kann seine eigene Widerspruchsfreiheit beweisen.

Die **Gödelschen Sätze** gehören mit Sicherheit zu den wichtigsten mathematischen und philosophischen Ergebnissen des 20.Jahrhunderts.

Sie beschreiben explizit in endgültiger, eindeutiger Form die unumstößlichen Grenzen des Algorithmierens und der Mathematik - und damit implizit auch die Grenzen des Computers und letztlich eine der grundlegenden Grenzen der menschlichen Erkenntnis.

Der Dichter und Schriftsteller H. M. Enzensberger hat sich von den Gödelschen Resultaten zu dem folgenden Gedicht anregen lassen:

Hommage à Gödel

Münchhausens Theorem, Pferd, Sumpf und Schopf,
ist bezaubernd; aber vergiß nicht,
Münchhausen war ein Lügner.

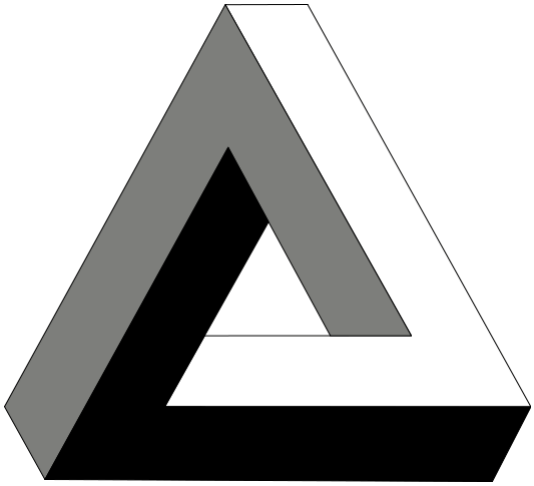
Gödels Theorem wirkt auf den ersten Blick
etwas unscheinbar, doch bedenk:
Gödel hat recht.

(H.M.Enzensberger, Gedichte 1955 - 1970, suhrkamp taschenbuch 4, 1971)

Interessantes Buch über die Theorien Gödels:
Douglas R. Hofstadter: *Gödel, Escher, Bach. Ein Endloses Geflochtenes Band.* 18. Auflage. Klett-Cotta, 2008

Douglas Richard Hofstadter (* 15. Februar 1945 in New York City) ist ein US-amerikanischer Physiker, Informatiker und Kognitionswissenschaftler.

Seine Systematik verbindet das mathematische Werk Kurt Gödels mit den kunstvollen Illustrationen M. C. Eschers und der Musik Johann Sebastian Bachs. Diese schöpferischen Werke setzt er in Beziehung zur Informatik, wie selbstbezüglichen Computerprogrammen den sogenannten Quines und den Strukturen der DNA, mithin der Molekularbiologie.



Penrose-Dreieck
(M.C.Escher)

Beispiel zur Frage der Entscheidbarkeit: **Goldbachsche Vermutung**

Die **Goldbachsche Vermutung** (nach Christian Goldbach,1690 - 1764) besagt, daß jede gerade Zahl größer als 2 (auf mindest eine Weise) als **Summe zweier Primzahlen** darstellbar ist.

z.B. 20 = 7 + 13 = 3 + 17

Die Frage, ob die Goldbachsche Vermutung aus den mathematischen Axiomen herleitbar oder widerlegbar ist, ist zur Zeit offen, dh. bis jetzt **nicht entscheidbar**.

Beispiel **Primzahlzwillinge**: Es existieren unendlich viele Zahlen **n** mit der Eigenschaft, dass **n** und **n+2** Primzahlen sind. Aussage wurde bis heute **nicht entschieden** (nicht bewiesen).

Wenn es für eine axiomatisch begründete Theorie stets richtig ist, daß sich jede beliebige im Rahmen einer Theorie formulierte Aussage entweder aus den Axiomen herleiten oder widerlegen läßt, dann nennt man die Theorie **entscheidbar**.

Bemerkungen:

- Es gibt (Programmier-) Sprachen, in denen nicht der Algorithmus sondern das Wissen über die Funktion notiert wird (→ Programmierparadigmen).

- Es gibt Ansätze, bei denen eine automatische Konstruktion eines Algorithmus für eine Funktion ***f*** anhand von endlich vielen Tupeln $(x_1, f(x_1)), (x_2, f(x_2)), ..., (x_n, f(x_n))$ versucht wird und man in Kauf nimmt, dass der so gewonnene Algorithmus eine ausreichend gute Näherung darstellt oder nicht in jedem Fall das korrekte Ergebnis liefert ($n>>1$) (→ Künstliche Intelligenz, Neuroinformationsverarbeitung, aber auch mathematische Approximation von Funktionen, z.B. Regressionsrechnung).

Unter **Korrektheit** versteht man in der Informatik die Eigenschaft eines Computerprogramms, einer Spezifikation zu genügen (siehe auch **Verifikation**).

Spezialgebiete der Informatik, die sich mit dieser Eigenschaft befassen, sind die **Formale Semantik** und die **Berechenbarkeitstheorie**.

Nicht abgedeckt vom Begriff ***Korrektheit*** ist, ob die ***Spezifikation*** die vom Programm zu lösende Aufgabe **korrekt** beschreibt (siehe dazu ***Validierung***).

Mit Testbeispielen allein kann in der Regel nicht die Korrektheit eines Programms nachgewiesen werden.

Der Nachweis der **Korrektheit** eines Programms kann nicht in allen Fällen geführt werden: das folgt aus dem ***Halteproblem*** bzw. aus dem ***Gödelschen Unvollständigkeitssatz***.

Auch wenn die **Korrektheit** für Programme, die bestimmten Einschränkungen unterliegen, bewiesen werden kann, so zählt die **Korrektheit** von Programmen allgemein zu den **nicht-berechenbaren** Problemen.

Das **Halteproblem** untersucht die Frage, ob es ein Verfahren (einen Algorithmus, ein Programm) gibt, mit dessen Hilfe man in endlich vielen Schritten entscheiden kann, ob ein beliebiges, vorliegendes Programm P bei Eingabe eines Satzes D von Eingabedaten stoppt, d.h. ob der Aufruf P(D) terminiert. Das Halteproblem ist nicht entscheidbar, d.h. es gibt keinen Algorithmus mit den oben genannten Eigenschaften.

Paradigmen des Programmierens und Fassungen des Algorithmen-Begriffs

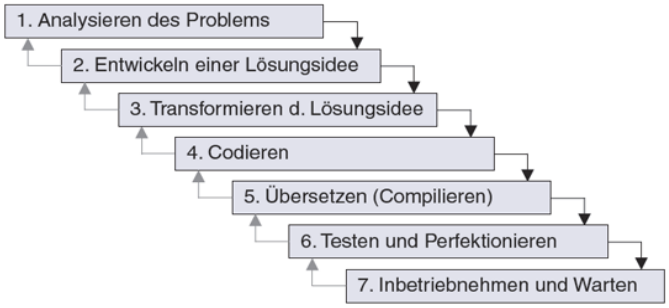
Die Paradigmen des Programmierens entsprechen recht genau den in der ersten Hälfte des 20. Jahrhunderts entwickelten gleichwertigen Fassungen des Algorithmenbegriffs ("Churchsche These").

Programmier-Paradigma	***Fassung des Algorithmus-Begriffs***
imperatives Programmieren	Turing Maschine (Pascal, C, Fortran, ...)
funktionales Programmieren	Lambda-Kalkül (Lisp, Haskell, Scheme, F#, ...)
regelbasiertes- und Logik-Programmierung	Prädikatenlogik, Prädikatenkalkül (Prolog)
objektorientiertes Programmieren	Typentheorie (C++, C#, Java, ...)

2.5 Programmierung als Teil der Softwareentwicklung

Der gesamte **Lebenszyklus** eines Softwareprodukts kann folgendermaßen in Phasen eingeteilt werden:

Problemanalyse
Spezifikation der Aufgabe
Grobentwurf
Projektierung, Entwurf des Datenmodells, Modularisierung, Feinentwurf
Algorithmieren (Programmieren, Codieren)
Prüfen, Verifizieren
Integration in Anwendungsumgebungen
Überführung in die Praxis
Nutzung
Wartung



Schrittweiser Prozess der Software-Entwicklung mit Rückkopplungen

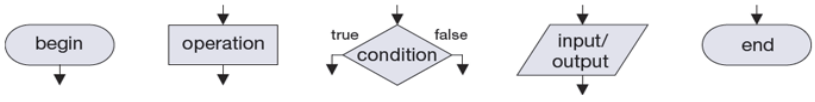
Besonderheiten der Software-Produktion:

. Ein Original genügt
. Kopien sind problemlos herstellbar.
. Kein physischer Verschleiß
. Rapider moralischer Verschleiß
. Elektronisch schnell zu versenden
. Schwieriger lizenzrechtlicher Schutz
. Abschottung bzw. Nachentwicklung der marktbeherrschenden Softwarefirmen
. Elektronische Ausspähung
. Hohe Loyalität der Mitarbeiter erforderlich

2.6 Flußdiagramm, Programmablaufplan

Grafische Notationen verwenden einfache grafische Symbole, vor allem Linien, Kreise, Rechtecke und Pfeile, um die Struktur und den Aufbau eines Algorithmus möglichst klar zu machen.

Flussdiagramm (Ablaufdiagramm) nach DIN 66001



Die wichtigsten Symbole für Ablaufdiagramme nach DIN 66001

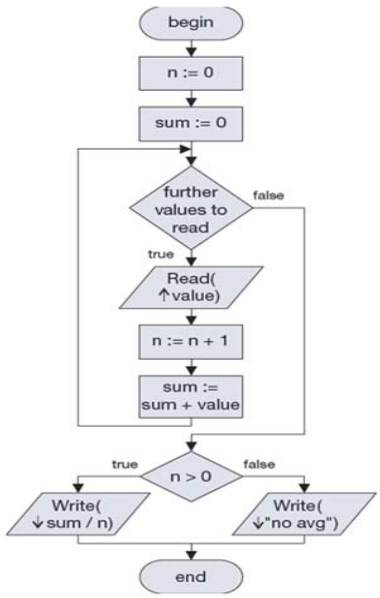


Abbildung Ablaufdiagramm nach DIN 66001 für einen Algorithmus zur Berechnung des arithmetischen Mittels

Der **Vorteil** dieser Darstellungsform ist, daß die Ablaufstruktur klar zutage tritt.

Der **Nachteil** liegt im hohen Darstellungs- und Änderungsaufwand.

In der Praxis wird häufig und in Abweichung von der DIN ein besser geeigneter Symbolvorrat verwendet:

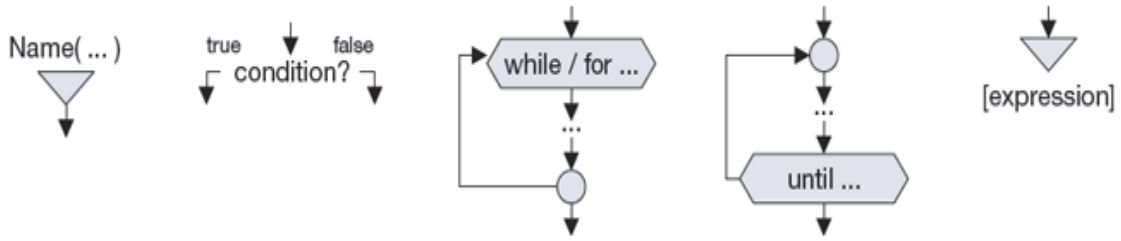
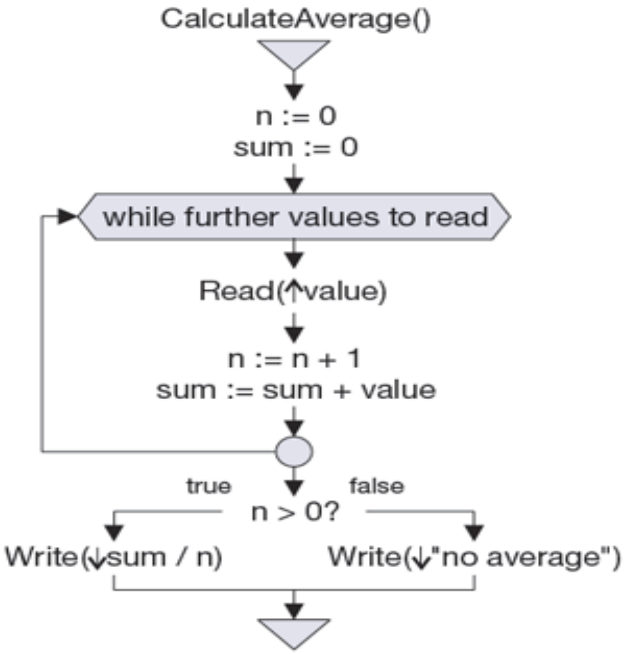
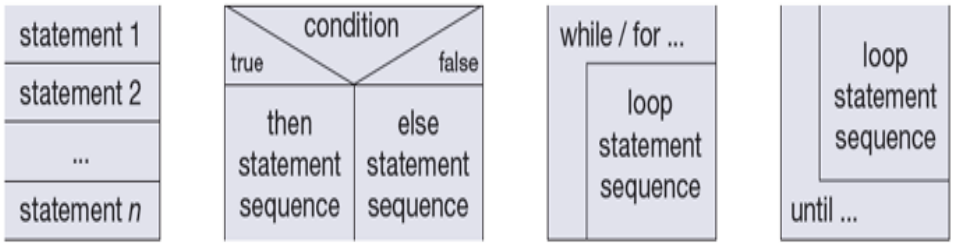


Abbildung Symbole für die reduzierte Notation von Ablaufdiagrammen



Ablaufdiagramm in reduzierter Notation für die Berechnung des Mittelwerts

Nassi-Shneiderman-Diagramm, (DIN 66262)



Die wichtigsten Symbole für Struktogramme

2.7 Struktogramm, Pseudocode

Ein **Nassi-Shneiderman-Diagramm** ist ein Diagrammtyp zur Darstellung von Programmentwürfen im Rahmen der Methode der strukturierten Programmierung. Er wurde 1972/73 von Isaac Nassi und Ben Shneiderman entwickelt und ist in der DIN 66261 genormt.

Da Nassi-Shneiderman-Diagramme Programmstrukturen darstellen, werden sie auch als **Struktogramme** bezeichnet.

Die Methode zerlegt das Gesamtproblem, das man mit dem gewünschten Algorithmus lösen will, in immer kleinere Teilprobleme – bis schließlich nur noch elementare Grundstrukturen wie *Sequenzen* und *Kontrollstrukturen* zur Lösung des Problems übrig bleiben. Diese können dann durch ein *Nassi-Shneiderman-Diagramm* oder ein *Flußdiagramm (Programmablaufplan-PAP)* visualisiert werden.

Die Vorgehensweise entspricht der sogenannten *Top-down-Programmierung*, in der zunächst ein Gesamtkonzept entwickelt wird, das dann durch eine Verfeinerung der Strukturen des Gesamtkonzeptes aufgelöst wird.

Böhm und Jacopini hatten 1966 nachgewiesen, dass sich jeder beliebige Algorithmus ohne unbedingte Sprunganweisung (GOTO) formulieren lässt.

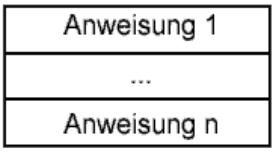
In Entwürfen mit Struktogramm bzw. Pseudocode sind beliebige Sprünge nicht abbildbar. Es gibt nur die strukturierten Konstrukte Sequenz, Schleife, Verzweigung und Funktionsaufruf.

Für Nassi-Shneiderman-Diagramme lassen sich trivial die Kontrollstrukturen moderner Programmiersprachen finden; für Flußdiagramme kann dies wesentlich schwieriger sein.

Sinnbilder nach DIN 66261

Die meisten der nachfolgenden Strukturblöcke können ineinander geschachtelt werden. Das aus den unterschiedlichen Strukturblöcken zusammengesetzte Struktogramm ist im Ganzen rechteckig, also genauso breit wie sein breitester Strukturblock.
Der Code bzw. Pseudocode entspricht der C/C++ - Notation

Linearer Ablauf (Sequenz) **Code (C/C++)**



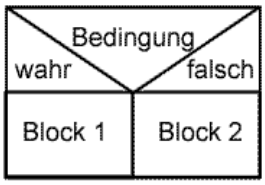

```
int i = 10;                 /* Anweisung 1 */
int summe = i + 5;          /* Anweisung 2 */
cout << summe << endl;      /* Anweisung 3 */
```

Jede Anweisung wird in einen rechteckigen Strukturblock geschrieben.

Die Strukturblöcke werden nacheinander von oben nach unten durchlaufen.

Leere Strukturblöcke sind nur in Verzweigungen zulässig.

Einfache Verzweigung



Code (C/C++)

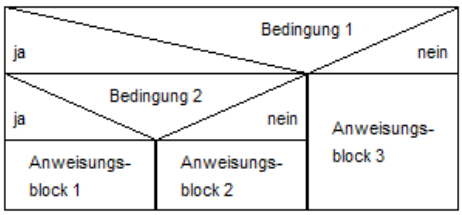
```
if( /* Bedingung */ )
{
      /* Block 1 */
}
else
                              {
      /* Block 2 */
}
```

Alternativ: bedingte Verarbeitung, Selektion, einfache Selektion

Nur wenn die Bedingung zutreffend (wahr) ist, wird Block 1 durchlaufen. Ein Anweisungsblock kann aus keiner, einer oder mehreren Anweisungen bestehen. Trifft die Bedingung nicht zu (falsch), wird Block 2 durchlaufen (Austritt unten).

Block 2 kann auch leer sein, d.h. im Code kann **else { /* Block 2 */ }** weggelassen werden.

Verschachtelte Auswahl

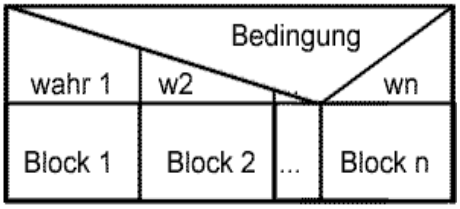


Code (C/C++)

```
 if( /* Bedingung 1*/ )
       if( /* Bedingung 2 */ )
       {
            /* Block 1 */
       }
       else
       {
           /* Block 2 */
      }
else  {
      /* Block 3 */
}
```

Es folgt eine weitere Bedingung. Die Verschachtelung ist ebenso im Nein-Fall möglich.

Fallauswahl (Mehrfache Auswahl)



Code (C/C++)

```
switch( i )
{
       case  1:    /* Block 1 */
                     break;
       case  2:    /* Block 2 */
                     break;
/*     case  ...:      Block ...
                    break; */
       default:    /* Block n */
}
```

Besonders bei mehr als drei abzuprüfenden Bedingungen geeignet: Der Wert von "Variable" kann bedingt auf Gleichheit wie auch auf Bereiche (größer/kleiner bei Zahlen) geprüft werden und der entsprechend zutreffende "Fall" mit dem zugehörigen Anweisungsblock wird durchlaufen. Eine Fallauswahl kann stets in eine verschachtelte Auswahl umgewandelt werden.

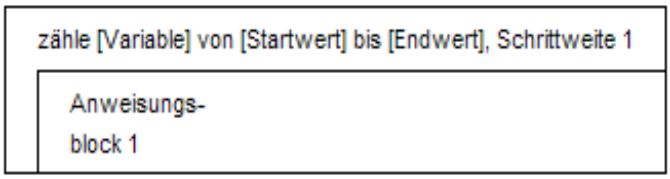
Wiederholung (Iteration)

Folgende drei Wiederholungen werden im Rahmen der Struktogramme unterstützt.

Bei falsch formulierter Endebedingung werden Iterationen **unendlich** oft abgearbeitet, d.h. der Algorithmus terminiert nicht.

Im Code kann mittels **break**-Anweisung die Schleife verlassen werden, mittels **continue**-Anweisung wird hinter die letzte Anweisung von block 1 gesprungen.

Zählergesteuerte Schleife



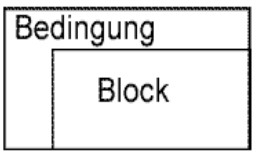
Code (C++)

```
for ( int i = start; i <= end; i=i+step)
{
        /* block 1 */
}
```

Wiederholungsstruktur, bei der die Anzahl der Durchläufe festgelegt ist. Als Bedingung muss eine Zählvariable **i** angegeben und mit einem Startwert **start** initialisiert werden. Ebenso muss ein Endwert **end** und die (Zähl-)Schrittweite **step** angegeben werden. Nach jedem Durchlauf des Schleifenkörpers (**block 1**) wird die Zählvariable **i** um die Schrittweite **step** inkrementiert (bzw. bei negativer Schrittweite dekrementiert) und mit dem Endwert **end** verglichen (Endebedingung). Ist die Endebedingung erfüllt, wird die Schleife verlassen, wenn nicht, wird der block 1 mit neuem i-Wert abgearbeitet. Die Laufvariable i darf nicht in **block 1** verändert werden.

Die Endebedingung kann auch alternativ formuliert werden, z.B. **i < end** . Das ist der Fall, wenn Vektoren mit **end** Elementen durchlaufen werden, wobei mit Index **0** begonnen wird.

Wiederholung mit vorausgehender Bedingungsprüfung (kopfgesteuerte Schleife, abweisende Schleife)



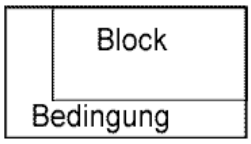
Code (C/C++)

```
while( /* Bedingung */ )
{
        /* Block*/
}
```

Der Schleifenkörperer (Block) wird nur durchlaufen, wenn (und solange) die Bedingung zutreffend (wahr) ist. Bereits der Versuch eines ersten Durchlaufs kann abgewiesen werden.

Diese Symbolik wird auch für die Zählergesteuerte Schleife (Anzahl der Durchläufe bekannt) benutzt, dabei muß jedoch die Zählervariable im Block explizit inkrementiert oder dekrementiert werden. Die kopfgesteuerte Schleife kann die zählergesteuerte Schleife ersetzen.

Wiederholung mit nachfolgender Bedingungsprüfung (fußgesteuerte Schleife, nicht abweisende Schleife)

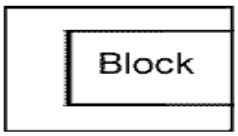


Code (C/C++)

```
do {
                /* Block */
} while( /* Bedingung */ );
```

Wiederholungsstruktur mit nachfolgender Bedingungsprüfung für den Abbruch. Der Schleifenkörper (Block) wird mindestens einmal durchlaufen, auch wenn die Bedingung von Anfang an nicht zutreffend (falsch) war.

Wiederholung ohne Bedingungsprüfung (Endlosschleife)



Code (C/C++)

```
while( true )
{
                /* Block */
}
```

Kann allenfalls durch einen Abbruch (**break**) bzw. durch das Verlassen der umfassenden Funktion mittels **return** verlassen werden.

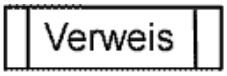
Abbruch

Code (C/C++)

```
break;
continue;
return;  bzw. return wert;
```

Der Abbruch stellt die Beendigung eines Programmteils dar

Aufruf (Unterverarbeitung)



Code

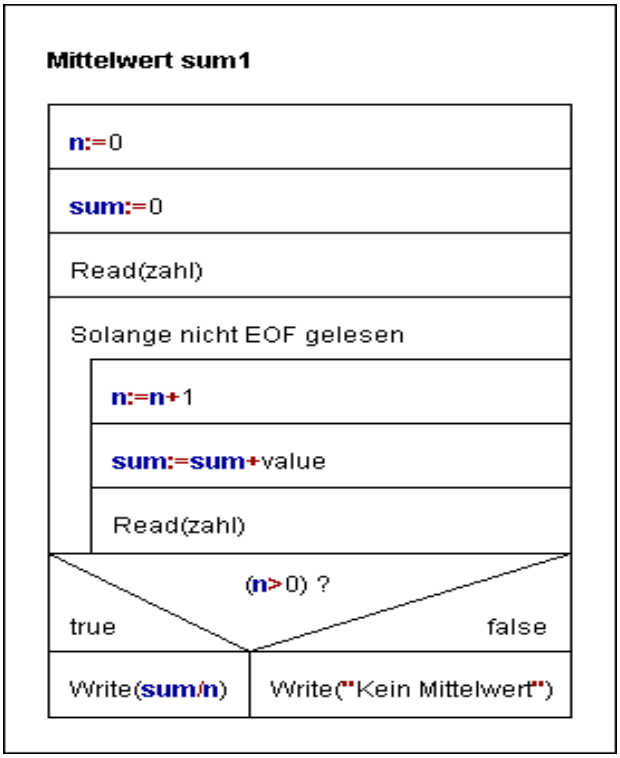
```
verweisname( /* parameterliste */ );
```

Symbol für den Aufruf eines Unterprogramms bzw. einer Prozedur oder Funktion. Nach deren Durchlauf wird zu der aufrufenden Stelle zurückgesprungen und der nächstfolgende Strukturblock durchlaufen.

Die Kommunikation der aufgerufenen Prozedur/Funktion mit dem aufrufenden Programm soll vollständig über eine Parameterliste und dem Funktionswert geschehen.

Parameter können *Eingabe-*, *Ausgabe-* oder *Ein-Ausgabeparameter* sein, wobei Eingabeparameter (↓) nur der Kommunikation von außen nach innerhalb und Ausgabeparameter (↑) von innen nach außerhalb der Prozedur/Funktion dienen. Ein-Ausgabeparameter (↓ ↑) dienen der Datenübermittlung in beide Richtungen. Im Unterschied zu einer Prozedur kann eine Funktion einen Wert oder eine Referenz auch über ihren Namen zurückgeben, z.B. double x = exp(2.0);

Struktogramm zum Beispiel zur Berechnung des **Mittelwertes** von unbestimmt vielem Zahlen:



C - Code Berechnung des **Mittelwertes** von **unbestimmt vielen Zahlen** (vgl. das folgende **sum1c.c**). Das Ende der Eingabe wird mit den Tasten **F6** bzw. **Strg+Z** realisiert, was das Dateiende (**EOF**) der Tastatureingabe darstellt.

```
#include <stdio.h>
// Mittelwertberechnung sum1c.c in C
//
// Beim ersten nicht in eine Zahl umwandelbaren Zeichen bricht die Eingabe ab,
// wobei die restlichen Zeichen bis zur Enter-Tasteneingabe uebersprungen werden.

void main(){
        int n = 0, i = 0;       // Anzahl Zahlen
        double sum = 0.0, value = 0.0;
        printf("\nZahl eingeben = ");
        i=scanf("%lf",&value);   // Zahl einlesen bzw. Dateiende (F6, Strg+Z)
        while(getchar() != '\n');

        while(i!=EOF){          // Zahl eingelesen (kein Datenende)?
             if(i){
                  n = n+1;              // Anzahl Zahlen inkrementieren
                  sum = sum + value;    // Zahl zu Summe addieren
             }
             printf("\nZahl eingeben = ");
             i=scanf("%lf",&value);      // Zahl einlesen bzw. Dateiende (F6, Strg+Z)
             while(getchar() != '\n');   // Eingabefehler ignorieren
        }
        if(n>0)                         // Anzahl > 0 ?
            printf("n = %d Zahlen, Mittelwert = %lf\n",n,sum/n);
        else
            printf("n = %d Zahlen, kein Mittelwert\n",n);
        getchar();
}
```

C - Code Berechnung des **Mittelwertes** von unbestimmt vielen Zahlen in **sum2c.c** unter Nutzung der *Funktion* **double_in2** zur robusten und korrekten Eingabe von double - Zahlen:

```
#include <stdio.h>
#define N 128UL
/* sum2c.c */
/* Funktion zur sicheren Eingabe einer double-Zahl. Beim ersten nicht
   umwandelbares Zeichen wird in sscanf weiter gesprungen, jedoch erneute
   Eingabe, wenn alle Zeichen nicht umwandelbar sind in sscanf,
   z.B. wird bei 3gg die 3 verwendet und gg uebersprungen */
/* int *eof - 1 bei EOF, sonst 0                */
/* char *s - Ausschrift Variableneingabe        */
/* char *p - Zeichenkettenvektor p[0] .. p[n-1] */
/* unsigned long n - Anzahl der Elemente von p  */

double double_in2(int *eof, char *s, char *p, unsigned long n){
       double d = 0.0; int i=0;
       char *c = 0;
       *eof = 0;
       if(!p){
                printf("Puffer existiert nicht\n"); return 0.0;
       }
       if(n<2UL){
                printf("Puffer zu kurz\n"); return 0.0;
       }
       while(1){
            if(s) printf(s);
            c = fgets(p,n,stdin);
            if(feof(stdin)){ *eof=1; return 0.0; }
            if(!c){
                 printf("Error, erneute Eingabe\n"); continue;
            }
            i = sscanf(p,"%lf",&d); while(getchar() != '\n'); /* 3g */

            if(!i || i==EOF){ printf("Error\n"); continue; }
            return d;
       }
}

void main(){
       int n = 0, i = 0;                      // Anzahl Zahlen
       double sum = 0.0, value = 0.0;
       char puffer[N]="";

       do {
            value = double_in2(&i, "Zahl eingeben = ", puffer, N);
            if(!i){
                 n = n+1;                    // Anzahl Zahlen inkrementieren
                 sum = sum + value;          // Zahl zu Summe addieren
            }
       } while(!i);

       if(n>0)                              // Anzahl > 0 ?
           printf("n = %d Zahlen, Mittelwert = %lf\n",n,sum/n);
       else
            printf("n = %d Zahlen, kein Mittelwert\n",n);
       getchar();
}
```

C - Code Berechnung des **Mittelwertes** von unbestimmt vielen Zahlen in **sum2c2.c** unter Nutzung der *Funktion* **double_in1** zur robusten und korrekten Eingabe von double - Zahlen:

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#define N 128UL

/* 1.Funktion zur sicheren Eingabe einer double-Zahl */
/* Ende beim ersten nicht umwandelbaren Zeichen */
/* char *s - Ausschrift Variableneingabe */
/* char *p - Zeichenkettenvektor p[0] .. p[n-1] */
/* unsigned long n - Anzahl der Elemente von p */
double double_in1(int *eof, char *s, char *p, unsigned long n){
     double d = 0.0;
     char *c = 0;
     *eof = 0;
     if(!p){
          printf("Puffer existiert nicht\n"); return 0.0;
     }
     if(n<2UL){
          printf("Puffer zu kurz\n"); return 0.0;
     }
     while(1){
          if(s) printf(s);
          c = fgets(p,n,stdin);
          if(feof(stdin)){
               *eof=1; return 0.0;
          }
          if(!c){ printf("Error, erneute Eingabe\n"); continue; }
          if((strlen(p)>=1) && (p[strlen(p)-1]=='\n'))
               p[strlen(p)-1]='\0';
          if(!strcmp(p,"0") || !strcmp(p,"0.0")) return 0.0;
          if(d = atof(p)) return d;
          printf("Error, erneute Eingabe\n");
     }
}

int  main(){
     int n = 0, i = 0;  // Anzahl Zahlen
     double sum = 0.0, value = 0.0;
     char puffer[N]="";

     do {
          value = double_in1(&i, "Zahl eingeben = ", puffer, N);
          if(!i){
               n = n+1;                    // Anzahl Zahlen inkrementieren
               sum = sum + value;          // Zahl zu Summe addieren
          }
     } while(!i);

     if(n>0)                // Anzahl > 0 ?
          printf("n = %d Zahlen, Mittelwert = %lf\n",n,sum/n);
     else
          printf("n = %d Zahlen, kein Mittelwert\n",n);
     getchar();
     return 0
}
```

C++ - Code Berechnung des **Mittelwertes** von unbestimmt vielem Zahlen (vgl. **sum1.c**). Das Ende der Eingabe wird mit den Tasten **F6** bzw. **Strg+Z** realisiert, was das Dateiende (EOF) der Tastatureingabe darstellt.

```
#include <iostream>
using namespace std;                    // sum1.cpp

// Mittelwertberechnung in C++

// Achtung: Beim ersten nicht in eine Zahl umwandelbaren Zeichen bricht die
// Eingabe ab, wobei die restlichen Zeichen bis zur Enter-Tasteneingabe
// uebersprungen werden. Wird bereits das erste Zeichen nicht umgewandelt,
// dann behaelt value seinen bisherigen Wert, was zu einem fehlerhaften
// Mittelwert fuehrt

void main(){
     int n = 0;          //Anzahl Zahlen
     double sum = 0.0, value = 0.0;
     cout<<"Zahl eingeben = ";
     cin>>value;         // Zahl einlesen bzw. Dateiende (F6, Strg+Z)

     while(!cin.eof()){      // Zahl eingelesen (kein Datenende)?
          cin.clear();       // Fehlerstatus cin zurücksetzen
          cin.ignore(INT_MAX, '\n');      // Eingabefehler ignorieren
          n = n+1;                         // Anzahl Zahlen inkrementieren
          sum = sum + value;               // Zahl zu Summe addieren
          cout<<"Zahl eingeben = ";
          cin>>value;    // Zahl einlesen bzw. Dateiende (F6, Strg+Z)
     }

     if(n>0)                        // Anzahl > 0 ?
         cout<<"n = "<<n
             <<" Zahlen, Mittelwert = "
             <<sum/n<<endl;       // Mittelwert berechnen und ausgaben
     else                          // Division durch 0 verhindern
         cout<<"n = "<<n
             <<" Zahlen, kein Mittelwert"<<endl;
     cin.clear();
     cin.get();
}

/*
Zahl eingeben = 4
Zahl eingeben = 5
Zahl eingeben = 6
Zahl eingeben = ^Z
n = 3 Zahlen, Mittelwert = 5
*/
```

C++ - Code Berechnung des **Mittelwertes** von unbestimmt vielen Zahlen (vgl. **sum2.cpp**). Das Ende der Eingabe wird mit den Tasten F6 bzw. Strg+Z realisiert, was das Dateiende (EOF) der Tastatureingabe darstellt. Eingabe ist robust und im Gegensatz zu **sum1.cpp** sicher.

```
#include <iostream>
using namespace std;
// Mittelwertberechnung      sum2.cpp
// Alle Fehleingabe werden abgefangen

double rdin(bool &, char *);      // Prototyp-Deklaration

void main(){
     int n = 0;                   //Anzahl Zahlen
     double sum = 0.0, value = 0.0;
     bool eof = false;

     while(true){                 // Endlosschleife
          value = rdin(eof, "Zahl eingeben = ");
          if(eof) break;          // Verlasse Schleife bei EOF
          n = n+1;                // Anzahl Zahlen inkrementieren
          sum = sum + value;      // Zahl zu Summe addieren
     }

     if(n>0)                      // Anzahl > 0 ?
         cout<<"n = "<<n
             <<" Zahlen, Mittelwert = "
             <<sum/n<<endl;       // Mittelwert berechnen und ausgeben
     else                         // Division durch 0 verhindern
         cout<<"n = "<<n
             <<" Zahlen, kein Mittelwert"<<endl;
}

double rdin(bool &eof, char * s = "double - Wert = "){
     double d = 0.0;
     char c = '\n';
     eof = false;
     do {
          // bei Fehler: cin.rdstate()!=0
          if(cin.rdstate() || c != '\n'){
               cout<<"Fehler, Eingabe wiederholen\n";
               cin.clear();
               cin.ignore(INT_MAX,'\n');
          }
          cout<<s; cin>>d;             // Einlesen
          if(cin.eof()){
               eof = true;             // EOF erkannt
               return d;               // 0.0 zurueck
          }
          c = cin.peek();              // naechstes Zeichen ohne zu Lesen
     } while(cin.rdstate() || c != '\n');
     return d;
}
/* Zahl eingeben = 4
Zahl eingeben = 5
Zahl eingeben = 6
Zahl eingeben = ^Z
n = 3 Zahlen, Mittelwert = 5 */
```