Was ist ein Algorithmus?

Software Entwicklung 1

Annette Bieniusa, Mathias Weber, Peter Zeller

In diesem Abschnitt beschäftigen wir uns dem Begriff des Algorithmus', der in der Informatik eine zentrale Rolle einnimmt. Dabei werden wir auf wichtige Konzepte wie Terminierung, Variablen, Ausführungszuständen, Aktionen und Zustandsänderung eingehen und den Unterschied zwischen Determinismus und Determiniertheit von Algorithmen kennenlernen.

1 Algorithmus

Die imperative Modellierung und Programmierung baut auf den klassischen Algorithmusbegriff auf. Eine Berechnung dabei wird als *zustandsändernder* Ablauf betrachtet. Damit orientiert sie sich am Berechnungskonzept von Rechnern, das auch auf der Beschreibung von Abläufen basiert, in denen sich in jedem Schritt der Ausführungszustand ändern kann. [1]
Es gibt viele Definitionen für den Begriff "Algorithmus", die sich aber in den wesentlichen Aspekten gleichen.

Definition Ein ***Algorithmus*** ist ein Verfahren zur schrittweisen ***Ausführung*** von (***Berechnungs-***) ***Abläufen***, das sich präzise und endlich beschreiben lässt, so dass:

- die Beschreibung auf wohlverstandenen, ausführbaren ("effektiven") Einzelschritten basiert;
- in jedem Schritt eine oder mehrere Aktionen (ggf. parallel) ausgeführt werden;
- jede Aktion von einem Zustand in einen Nachfolgezustand führt.

Man sagt, die Ausführung eines Algorithmus ***terminiert***, wenn sie nach endlich vielen Schritten beendet ist; andernfalls spricht man von einer ***nicht-terminierenden*** Ausführung.

[1] Neben dem *imperativen* Programmierparadima, das unter anderem Java, C/C++, JavaScript oder Python zugrunde liegt, gibt das deklarative Programmierparadigma. Dabei wird von den einzelnen Verarbeitungsschritten abstrahiert. Programme in deklarativen Sprachen beschreiben den Zusammenhang zwischen Eingabe und Ausgabe, nicht aber, wie die Ausgabe anhand der Eingabe konkret berechnet werden soll.

Algorithmen lassen sich mit unterschiedlichen Sprachmitteln beschreiben:

- umgangssprachlich,
- mit mathematischer Sprache,
- in graphischer Notation, oder mit
- mit programmiersprachlichen Mitteln.

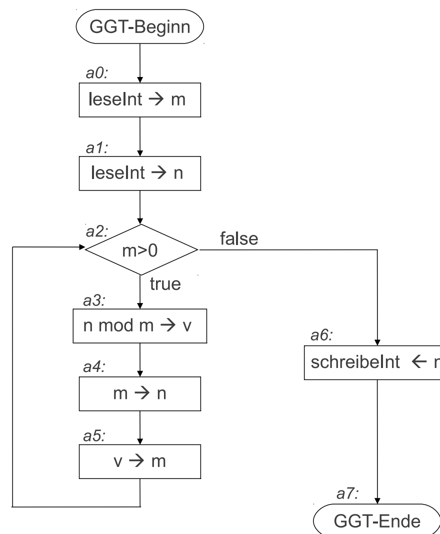
Ein Algorithmus ist dabei *unabhängig* von der verwendeten Beschreibungstechnik bzw. Sprache.

1.1 Beispiel: Algorithmus zur Berechnung des größten gemeinsamen Teilers

Betrachten wir als Beispiel einen Algorithmus zur Berechnung des größten gemeinsamen Teiles (ggT) zweier ganzer Zahlen. Zunächst die Formulierung in Umgangssprache:

> Seien m, n, v Variablen für Integer-Werte.
> Lese die Werte $w1$ und $w2$ ein, für die der ggT berechnet werden soll und weise $w1$ an m und $w2$ an n zu. Solange der Wert von m größer als 0 ist, führe die folgenden drei Schritte aus und prüfe danach wieder die Bedingung:
>
> - Berechne $n \bmod m$ und weise das Ergebnis an v zu;
> - Weise den Wert von m an n zu;
> - Weise den Wert von v an m zu;
>
> Gebe den Wert aus, den n enthält.

Dieser Algorithmus kann auch in graphischer Notation als Flussdiagramm formuliert werden:



Als Java-Programm sieht der Algorithmus folgendermaßen aus:

```
public class GGT { // Berechnet ggT fuer 2 gelesene Werte
 public static void main(String[] args){
   int m = Integer.parseInt(args[0]);
   int n = Integer.parseInt(args[1]);

   while (m > 0) {
     int v = n % m;
     n = m;
     m = v;
   }
   System.out.println("ggT:" + n);
 }
```

1.2 Variablen und Zustände

Um die Zustände zwischen den Schritten präziser fassen zu können, werden in Programmen in der Regel Variablen verwendet. Wir erinnern uns:

Definition Eine ***Speichervariable*** (oder einfach nur: Variable) ist ein Speicher/-Behälter für Werte. Charakteristische Operationen auf einer Variablen v:

- ***Zuweisen*** eines Werts w an v;
- ***Lesen*** des Wertes, den v enthält/speichert/hat.

Der Zustand einer Variablen v ist undefiniert, wenn ihr noch kein Wert zugewiesen wurde; andernfalls ist der Zustand von v durch den gespeicherten Wert charakterisiert.
Jeder Schritt bei der Ausführung eines Algorithmus führt von einem ***Ausführungszustand*** zum ***Nachfolgezustand***.
Ein Ausführungszustand ist gekennzeichnet durch

- den ***Speicherzustand*** (im Wesentlichen der Zustand der Variablen);
- den ***Steuerungszustand*** (vereinfacht gesagt, die Stelle im Programm, an der die Ausführung angekommen ist).

Ein Ausführungsschritt führt zu einer Zustandsänderung, also einer Veränderung von Speicher- und/oder Steuerungszustand.
In einem Ausführungsschritt wird üblicherweise eine ***Aktion*** ausgeführt. Aktionen sind insbesondere:

- Zuweisungen an Variablen
- Prüfen von Bedingungen
- Kommunikation mit der Umgebung (Ein- und Ausgabe)

Die Aktion bestimmt nachfolgende Steuerungszustände bzw. die Terminierung des Algorithmus.
Der ***Ablauf*** eines Algorithmus zu gegebenen Eingaben wird charakterisiert durch

- die Sequenz der Ausführungszustände und
- die Sequenz der ausgeführten Aktionen.

Definition Ein Algorithmus heißt ***deterministisch***, wenn für alle Eingabedaten der Ablauf des Algorithmus eindeutig bestimmt ist. Andernfalls heißt er ***nicht-deterministisch***.

Beispiel Betrachten wir folgende Aufgabenstellung:

Erkenne, ob eine Eingabezeichenreihe über Kleinbuchstaben eines der Worte "hallo", "hi" oder "salut" enthält.

Die Eingabe ist gegeben als eine Eingabezeichenreihe über Kleinbuchstaben; die möglichen Ausgaben sind "ja" (falls die Eingabe einer der genannten Zeichenfolgen enthält) bzw. "nein" (falls dies nicht der Fall ist).
Der folgende nicht-deterministischer Algorithmus löst die Aufgabenstellung. Dabei bezeichnet z die Eingabe und prefixlen eine Variable, welche die Menge der Positionen enthält, an denen wir in z nach den Worten "hallo", "hi" oder "salut" suchen werden.

- Wir initialisieren zunächst prefixlen $:= \{\ 0, \ldots, \text{laenge}(z) - 2\ \}$
- Solange prefixlen $\neq \varnothing$, tue folgendes:
  - Wähle ein x aus prefixlen aus;
  - prefixlen := prefixlen $\setminus \{x\}$;
  - w := "z ohne die ersten x Buchstaben";
  - Prüfe, ob w mit "hallo", "hi" oder "salut" beginnt;

  - Falls ja, terminiert der Algorithmus mit "ja"; andernfalls fahre mit dem nächsten Durchlauf der Schleife fort.

- Terminiere mit der Ausgabe "nein".

Frage 1: Wieso ist der Algorithmus nicht-deterministisch?

Eine weitere wichtige Eigenschaft von Algorithmen ist die Determiniertheit, der sich wesentlich vom Determinismus unterscheidet.

Definition Ein Algorithmus heißt ***determiniert***, wenn er bei gleichen zulässigen Eingabewerten stets das gleiche Ergebnis liefert.
Andernfalls heißt er ***nicht-determiniert***.

Aus der Definition lässt sich ableiten, dass jeder Algorithmus, der eine mathematische Funktion berechnet, determiniert ist, da eine Funktion jedem Element der Definitionsmenge genau ein Element der Zielmenge zuordnet.
Auch der nicht-deterministische Algorithmus des obigen Beispiels ist determiniert: Die Reihenfolge, in der die Elemente von prefixlen überprüft werden, ist nicht festgelegt. Das Ergebnis selbst wird aber nicht von der Reihenfolge beeinflusst.

2 Anweisungen in Java

Anweisungen (engl. *statements*) sind wichtige programmiersprachliche Beschreibungsmittel in Programmiersprachen, die Programme in Form von Zustandsänderungen formulieren.

- ***Einfache*** Anweisungen beschreiben Aktionen.
- ***Zusammengesetzte*** Anweisungen beschreiben, wie mehrere Aktionen auszuführen sind.

In diesem Abschnitt betrachten wir die folgenden Arten von Anweisungen:

- Einfache Anweisungen (Zuweisung, Prozeduraufruf [→ nächste Vorlesung])
- Anweisungsblöcke
- Schleifenanweisungen (while-, do-, for-Anweisungen)
- Verzweigungsanweisungen:
  - bedingte Anweisung
  - Fallunterscheidung
  - Auswahlanweisung
- Sprunganweisungen (Abbruchanweisung)

Diese Anweisungen sind finden sich in den meisten imperativen Sprachen. Wir geben für jeden Anweisungstyp die Syntax in Java an und diskutieren die Semantik. Für eine detaillierte Diskussion verweisen wir auf:

Kapitel 1.3 aus R. Sedgewick, K. Wayne: Einführung in die Programmierung mit Java. 2011, Pearson Studium.

2.1 Variablendeklaration und -zuweisung

- Variablendeklaration

  `Deklaration` $\rightarrow$
  `Typ` $\ll$ *Bezeichner* $\gg$ `;`
  | `Typ` $\ll$ *Bezeichner* $\gg$ `=` `Ausdruck;`

  `Typ` $\rightarrow$ `PrimitiverTyp` | $\ll$ *Bezeichner* $\gg$

  `PrimitiverTyp` $\rightarrow$
  `byte` | `short` | `int` | `long` | `float`
  | `double` | `char` | `boolean`

- Zuweisungen[2]

  `Zuweisung` $\rightarrow$ $\ll$ *Bezeichner* $\gg$ `=` `Ausdruck` `;`

In einer Zuweisung (und anderen Sprachkonstrukten) kann eine Variable `v` mit zwei Bedeutungen vorkommen.

$$a = 7 + a;$$

L-Wert ↗ (linkes a)   ↖ R-Wert (rechtes a)

1. Das Vorkommen links vom Zuweisungszeichen meint die Variable. Man spricht vom *L-**Wert*** (engl. *l-value*) des Ausdrucks v .

2. Das Vorkommen rechts vom Zuweisungszeichen meint den in v gespeicherten Wert. Man spricht vom *R-**Wert*** (engl. *r-value*) des Ausdrucks v.

Die Unterscheidung wird wichtiger, wenn auch links des Zuweisungszeichens komplexere Ausdrücke stehen.

[2]In Java ist eine Zuweisung syntaktisch ein Ausdruck, liefert also einen Wert und zwar das Ergebnis der Auswertung von der rechten Seite der Zuweisung.

2.2 Anweisungsblöcke

Ein ***Anweisungsblock (engl. block statement)*** ist eine Liste bestehend aus Deklarationen und Anweisungen.
Syntax in Java:

```
Anweisung → { DeklAnweisListe }

DeklAnweisListe → ε
                | Deklaration DeklAnweisListe
                | Anweisung DeklAnweisListe
```

Semantik:
Stelle den Speicherplatz für die Variablen bereit und führe die Anweisungen der Reihe nach aus.
Beispiel für einen Anweisungsblock:

```
{ int i; i = 7; int a; a = 27 % i; }
```

Üblicherweise schreibt man Deklarationen und Anweisungen untereinander, so dass ein Textblock entsteht. Anweisungen in einem Block werden in der Regel eingerückt, um die Lesbarkeit zu verbessern.

```
{
  int i;
  int x;
  i = 34;
  x = i * 5;
}
```

2.3 Kontrollfluss: Verzweigungen und Schleifen

Verzweigungsanweisungen (engl. *branch statements / conditionals*) stellen Bedingungen an die Ausführung einer Anweisung oder wählen einen von mehreren Zweigen zur Ausführung aus.
Wir betrachten hier zunächst zwei Arten von Verzweigungsanweisungen:

- Bedingte Anweisung
- Fallunterscheidung

2.3.1 Bedingte Anweisung

Syntax in Java:

```
Anweisung → if ( Ausdruck ) Anweisung
```

Semantik:
Werte den Ausdruck aus; er muss sich zu einem boolschen Wert ergeben. Ist das Ergebnis `true`, führe die Anweisung aus.

Beispiel:
Wir gehen in diesen Beispielen davon aus, dass die Integer-Variablen `x,y,z` bereits deklariert und initialisiert worden sind.

```
// Absolutwert einer Zahl
if (x < 0) { x = -x; }

// Testet, ob Divisor null ist, bevor die Division durchgefuehrt wird
if (x != 0) { z = y/x; }

// Sortieren von x und y: x soll kleiner als y sein
if (x > y) {
  int t = x;
  x = y;
  y = t;
}
```

2.3.2 Fallunterscheidung

Syntax in Java:

```
Anweisung →
    if ( Ausdruck )  Anweisung
    else Anweisung
```

Semantik:

Werte den Ausdruck aus; er muss sich zu einem boolschen Wert ergeben. Ist das Ergebnis `true`, führe die erste Anweisung aus, andernfalls führe die Anweisung nach `else` aus.

Beispiel: Fairer Münzwurf
Math.random() liefert eine Zufallszahl aus dem Bereich [0.0, 1.0).

```
if (Math.random() < 0.5) {
  System.out.println("Kopf");
} else {
  System.out.println("Zahl");
}
```

Beispiel: Maximum zweier Zahlen

```
// Maximum von x und y
int x, y;
int max;

x = ...;  // Initialisieren von x
y = ...;  // Initialisieren von y

if ( x > y ) {
  max = x;
} else {
  max = y;
}
```

Frage 2: Wie ermittelt man das Maximum dreier Zahlen (x,y,z)?

2.3.3 Schleifenanweisungen

Schleifenanweisungen (engl. *loop statements*) steuern die iterative, d.h. wiederholte Ausführung von Anweisungen/Anweisungsblöcken.
Wir betrachten hier die folgenden Schleifenanweisungen:

- while-Anweisung
- do-Anweisung
- Zählanweisung (for-Anweisung) ($\Rightarrow$ nächste Vorlesung)

while-Anweisung

Syntax in Java:

```
Anweisung →
    while( Ausdruck ) Anweisung
```

Semantik: Werte den Ausdruck (sogenannte ***Schleifenbedingung***) zu einem boolschen Wert aus. Ist die Bedingung erfüllt, führe die Anweisung aus, den sogenannten ***Schleifenrumpf***, und wiederhole den Vorgang. Andernfalls beende die Ausführung der Schleifenanweisung.
Beispiel: ggt
Schleife zur Berechnung des größten gemeinsamen Teilers:

```
while ( m > 0 ) {
    v = n % m;
    n = m;
    m = v;
}
```

Bemerkung:
Der Schleifenrumpf ist in den meisten Fällen ein Anweisungsblock. Syntaktisch korrekt ist aber auch z.B.:

```
while( true )  System.out.println(i);
```

Das Gleiche gilt für bedingte Anweisungen und Fallunterscheidungen. Um Fehler zu vermeiden sollten Sie immer die Variante mit einem Anweisungsblock verwenden.
Beispiel: Berechnung von Zweierpotenzen

```
/*
* Berechnet die Tabelle aller Zweierpotenzen,
* die kleiner als 2^n sind.
* Der Wert von n wird dabei als Befehlszeilenargument
* uebergeben.
*/

public class Zweierpotenz {
```

```
  public static void main(String[] args) {
    int n = Integer.parseInt(args[0]);
    int v = 1;
    int i = 0;

    while (i < n) {
      System.out.println(i + " " + v);
      v = 2 * v;
      i = i + 1;
    }
  }
}
```

Terminierung Durch Programmierfehler ergeben sich bisweilen Situationen, in denen eine Schleife unendlich oft ausgeführt wird ("Endlosschleife"). Ein Programmierer muss sich daher immer die folgende Frage stellen:

Wird die Ausführung der Schleife irgendwann abgebrochen?

Konkret müssen folgende Fragen beantwortet werden können:

1. Welche Werte müssen die Variablen in der Schleifenbedingung annehmen, damit diese nicht mehr erfüllt ist?
2. Wie verändern sich die Werte dieser Variablen in jedem Schleifendurchlauf?
3. Wird die Schleifenbedingung daher irgendwann nicht mehr erfüllt?

Als Hilfestellung kann man sich überlegen, welche Werte die Variablen annehmen, nachdem die Schleife das erste Mal, das zweite, etc., das letzte Mal durchlaufen wird.
Beispiel: Terminierung des Programms zur Berechnung der Zweierpotenz

1. Die Schleifenbedingung ist nicht mehr erfüllt, wenn die Variable `i` größer oder gleich der Eingabe `n` ist.
2. In jedem Schleifendurchlauf wird der Wert der Variablen `i` um 1 erhöht, der Wert von `n` verändert sich nicht.
3. Fallunterscheidung:
   - *1. Fall* `n` ≤ 0: Da Variable `i` mit 0 initialisiert wird, wird die Schleifenbedingung nie erfüllt, und der Schleifenrumpf nicht ausgeführt.
   - *2. Fall* `n` > 0: Da `i` mit 0 initialisiert wird und in jedem Schleifendurchlauf um 1 erhöht wird, gilt nach dem k-ten Schleifendurchlauf, dass `i` den Wert k hat. Nach dem n-ten Schleifendurchlauf hat `i` also den Wert n, und die Schleifenbedingung ist dann nicht mehr erfüllt.
4. Diese beiden Fälle decken alle möglichen Eingabewerte für `n` ab; daher terminiert die Ausführung für alle Eingaben.

do-Anweisung

Syntax in Java:

```
Anweisung →
  do Anweisung while( Ausdruck ) ;
```

Semantik: Führe die Anweisung aus.
Werte danach den Ausdruck zu einem boolschen Wert aus.
Wenn die Bedingung erfüllt ist, wiederhole den Vorgang.
Andernfalls beende die Ausführung der Schleifenanweisung.

Die do-Anweisung ist immer dann sinnvoll, wenn man einen Vorgang mindestens einmal ausführen muss. In solchen Situationen spart man sich gegenüber der Verwendung der while-Schleife die anfänglich unnötige Auswertung der Bedingung.

2.4 Sprung- und Auswahlanweisungen

Sprunganweisungen (engl. *jump statements*) legen eine Fortsetzungsstelle der Ausführung fest, die möglicherweise weit von der aktuellen Anweisung entfernt liegt.
Wir betrachten hier nur Sprünge, die der Programmstruktur folgen.

2.4.1 Abbruchanweisung

Syntax in Java:

```
Anweisung → break ;
```

Semantik: Die Ausführung wird mit der Anweisung fortgesetzt, die nach der umfassenden Schleife oder Auswahlanweisung kommt.

Typische Einsatzgebiete für die `break`-Anweisung:

1. In Auswahlanweisung: siehe unten.
2. In Schleifenanweisungen:

```
while( true ) {
    ...
    if( <Abbruchbedingung> ) break;
    ...
}
```

2.4.2 Auswahlanweisung

Syntax in Java:

```
Anweisung →
    switch ( Ausdruck ) { FallListe }
```

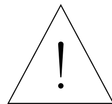
```
FallListe →
    Fall FallListe
  | ϵ

Fall →
    case Ausdruck : DeklAnweisListe
  | default : DeklAnweisListe
```

Auswahlanweisungen erlauben es, in Abhängigkeit vom Wert eines Ausdrucks direkt in einen von endlich vielen Fällen zu verzweigen.
In Java gibt es dafür die `switch`-Anweisung.
Beispiel:

```
public class SwitchExample {
  public static void main(String[] args) {
    String eingabe = args[0];
    switch( eingabe ) {
      case "a":  System.out.println("A wie Apfel");      break;
      case "b":  System.out.println("B wie Banane");     break;
      case "c":  System.out.println("C wie Clementine"); break;
      default:   System.out.println("Falsches Eingabezeichen");
    }
  }
}
```

Die Werte hinter `case` müssen hierbei konstante Ausdrücke sein; Funktionen und veränderbare Variablen dürfen hier also nicht verwendet werden. Jeder Fall darf höchstens einmal in der Anweisung auftauchen. Der Typ des Ausdrucks muss dabei ein primitiver Typ, ein Aufzählungstyp oder String sein. Das Verwenden des Datentyps `String` in Auswahlanweisungen ist in Java erst seit Version 1.7 möglich.
Die `switch`-Anweisung wertet zuerst den Ausdruck aus und springt dann zum Fall mit dem entsprechenden Wert. Falls kein entsprechender Fall existiert springt die Ausführung zum `default` Fall, falls dieser auch nicht existiert zum Ende der `switch`-Anweisung.



Die Ausführung der `switch`-Anweisung endet nicht, wenn der nachfolgende Fall beginnt. Es muss eine `break`-Anweisung verwendet werden, um die Ausführung der `switch`-Anweisung zu beenden.

Weitere Arten von Anweisungen werden im Verlauf der Vorlesung noch behandeln.

2.5 Unterschied: Anweisung/Ausdruck

Wir haben in der letzten Vorlesung bereits Ausdrücke kennengelernt.
Ein ***Ausdruck*** (engl. ***expression***) in Java ist (u.a.)

- ein Literal,
- ein Bezeichner (Variable, Name),
- die Anwendung einer Operation auf einen oder mehrere Ausdrücke, etc.

Ausdrücke sind das Sprachmittel zur Beschreibung von Werten. Die Auswertung von Ausdrücken liefert ein Ergebnis.
Im Gegensatz dazu verändert die Ausführung von Anweisungen den Zustand. Im Allgemeinen liefern sie kein Ergebnis. Anweisungen können Ausdrücke enthalten, z.B. zur Beschreibung von Werten, die Variablen zugewiesen oder auf der Konsole ausgegeben werden sollen.

Zusammenfassung: Neue Sprachelemente

```
Anweisung →
      { DeklAnweisListe }
    | Zuweisung
    | if ( Ausdruck ) Anweisung
    | if ( Ausdruck ) Anweisung else Anweisung
    | while( Ausdruck ) Anweisung
    | do Anweisung while( Ausdruck ) ;
    | break ;
    | switch ( Ausdruck ) { FallListe }

DeklAnweisListe →
      Deklaration DeklAnweisListe
    | Anweisung DeklAnweisListe
    | ϵ

Deklaration →
      Typ  ≪ Bezeichner ≫ ;
    | Typ  ≪ Bezeichner ≫  =  Ausdruck;

Typ → PrimitiverTyp | ≪ Bezeichner ≫

PrimitiverTyp →
      byte | short | int | long | float
    | double | char | boolean

Zuweisung → ≪ Bezeichner ≫  =  Ausdruck ;

FallListe →
      Fall FallListe
    | ϵ

Fall →
      case Ausdruck : DeklAnweisListe
    | default : DeklAnweisListe
```

Hinweise zu den Fragen

Hinweise zu Frage 1: Die Anweisung "Wähle ein x aus prefixlen aus" lässt die Auswahlreihenfolge offen. Der Ablauf des Algorithmus' ist daher nicht eindeutig bestimmt, sondern kann – bei gleicher Eingabe – in jedem Programmdurchlauf anders sein.

Hinweise zu Frage 2: Eine mögliche Implementierung ist die folgende:

```
int x, y;
int max;

x = ...;  // Initialisieren von x
y = ...;  // Initialisieren von y

if ( x > y ) {
  if ( x > z ) {
    max = x;
  } else {
    max = z;
  }
} else {
  if ( y > z ) {
    max = y;
  } else {
    max = z;
  }
}
```