Informatik I

1. ***Was ist Informatik?***
2. ***Vom Problem zur Lösung: Algorithmen***
3. ***BNF zur Beschreibung von Sprachen***
4. ***Aussagen- und Prädikatenlogik***
5. ***Korrektheit von Algorithmen***
6. ***Mengen, Relationen und Funktionen***
7. ***Berechenbarkeit***
8. ***Komplexität***
9. ***Rekursive Algorithmen***
10. ***Lösungsverfahren***
11. ***Parallele Algorithmen***
12. ***Programmierparadigmen***
13. ***Elementare Datentypen***
14. ***Komplexe Datentypen***
15. ***Abstrakte Datentypen***
16. ***Das objektorientierte Paradigma***

1. Was ist Informatik?

Informatik (Computer Science) ist die Wissenschaft von der systematischen Verarbeitung von Daten, besonders der automatischen Verarbeitung mit Hilfe von Digitalrechnern.

Das Kunstwort Informatik ist gebildet aus

Information

Automatik

Der Begriff „informatique“ wurde bereits von Descartes im 17. Jahrhundert eingeführt als „Behandung von Information mit rationalen Mitteln“:

Reduktion von Problemen auf bereits gelöste

Zerlegung von Problemen in Teilprobleme

Verifikation des Resultats

Methoden der Informatik:

Theorie (Mathematik)

- **Definition**
- **Theorem**
- **Beweis**

Abstraktion (Naturwissenschaften)

- **Modellhypothese**
- **Modell**
- **Experiment**
- **Analyse**

Design (Ingenieurwissenschaften)

- **Formulierung der Anforderungen**
- **Spezifikation**
- **Entwurf und Implementierung**
- **Verifikation und Validierung**

Informatik als Wissenschaft ist das systematische Studium von informationsbeschreibenden und –transformierenden Prozessen in Bezug auf deren

- **Theorie**
- **Analyse**
- **Entwurf**
- **Effizienz**
- **Implementierung**
- **Anwendung**

Teilgebiete der Informatik

Theoretische Informatik

- **Formale Methoden**
- **Automatentheorie**
- **Datenstrukturen (x)**
- **Algorithmen (x)**
- **Berechenbarkeit (x)**
- **Komplexität (x)**

Praktische Informatik

- **Betriebssysteme**
- **Programmiersprachen und Compiler**
- **Datenbanken**
- **Netzwerke**
- **Multimedia**
- **Künstliche Intelligenz**
- **Softwaretechnik**

Technische Informatik

- **Hardware**
- **Rechnerarchitektur**
- **Mikroprozessoren**

Angewandte Informatik

2. Vom Problem zur Lösung: Algorithmen

Vorgehensweise (ideal):

1. **Formale Erfassung der Problemstellung**
2. **Analyse der Problemstellung (Reduktion auf gelöste Probleme, Zerlegung in Teilprobleme)**
3. **Entwicklung eines Lösungsmodells**
4. **Entwurf eines Lösungsverfahrens (Algorithmus)**
5. **Nachweis der Korrektheit**
6. **Analyse der Effizienz / Komplexität**
7. **Implementierung**
8. **Test und Fehlerbeseitigung**
9. **Dokumentation**

Abweichungen in der Praxis:

Es gibt Probleme ohne Lösung.

Es gibt Probleme ohne effiziente Lösung.

Der Nachweis der Korrektheit wird häufig durch „Plausibilität“ geführt.

Es wird häufig iterativ vorgegangen.

Algorithmus:

Ein Algorithmus ist ein Verfahren mit einer präzisen, in einer genau festgelegten Sprache abgefassten Beschreibung unter Verwendung von effektiven (d.h., tatsächlich ausführbaren) Verarbeitungsschritten.

Eigenschaften:

Endliche Formulierung

Endliche Laufzeit

Endlicher Ressourcenverbrauch

Präzise Verarbeitungsschritte

Sprachen zur Formulierung von Algorithmen:

Natürliche Sprache

Programmiersprache

Diagramme

Pseudo Code

Sprachelemente:

Führe Schritte ... durch (Sequenz)

Falls die Bedingung ... erfüllt ist, führe Schritte ... durch:

Ansonsten führe Schritte ... durch (Selektion)

Solange die Bedingung ... erfüllt ist, führe Schritte ... durch (Iteration)

Anstelle der sequentiellen Durchführung von Verarbeitungsschritten in einer vorgegebenen Reihenfolge ist auch die Durchführung in beliebiger Reihenfolge bzw. die parallele Durchführung zu betrachten! Wir beschränken uns zunächst auf die klassische Sequenz.

Beispiele:

Kochrezepte

Berechnungsvorschriften

Bestimmung der Summe der ersten n natürlichen Zahlen:

1. **Setze Summe auf den Wert 0.**
2. **Setze i auf 1.**
3. **Solange i kleiner oder gleich n ist, führe Schritte 4 und 5 durch:**
4. **Setze Summe auf Summe+i**
5. **Setze i auf i+1**

3. BNF zur Beschreibung von Sprachen

Zu einer Sprachbeschreibung gehören:

Alphabet

Wörter

Syntax (Struktur)

Semantik (Bedeutung)

Regelsystem zur Erzeugung syntaktisch korrekter Sätze aus syntaktisch korrekten Sätzen und Grundbegriffen (Grammatik)

Backus-Naur-Form (BNF)

1959 von John Backus und Peter Naur für die Spezifikation von ALGOL-60 entwickelt.

BNF besteht aus

- **Terminalen (Zeichen des Alphabets, reservierte Wörter)**
- **Nonterminalen (syntaktische Begriffe)**
- **Einem ausgezeichneten Nonterminal, dem Startsymbol**
- **Einer Menge von Regeln der Form**
  Nonterminal ::= Folge von Terminalen und Nonterminalen
  <A>::=B wird wie folgt beschrieben:
  A kann durch B ersetzt werden
  A steht für B,
  B ist Alternative für A
  <A>::=B heißt Ableitungsschritt
- **Im Rahmen von Ableitungsschritten können | für Auswahlmöglichkeiten, [] für Optionalität und {} zur Zusammenfassung verwendet werden.**

Die Beschreibung einer BNF erfolgt i.a. rekursiv.

Anstelle einer BNF kann man auch ein Syntaxdiagramm verwenden.

Beispiel einer BNF:

Durch folgende BNF wird die Menge der natürlichen Zahlen inklusive Null beschrieben, wobei führende Nullen erlaubt sind.

<Zahl>::= <Ziffer> | <Ziffer><Zahl>
<Ziffer>::= 0|1|2|3|4|5|6|7|8|9

Alternativ kann der syntaktische Begriff <Zahl> wie folgt definiert werden:
<Zahl>::= <Ziffer> [<Zahl>]
<Zahl>::= <Ziffer> { <Ziffer>}

Ein Wort der Sprache ist eine Folge von Terminalen, die sich aus dem Startsymbol ableiten lässt. Es gibt zwei Varianten, Ableitungen zu beschreiben:

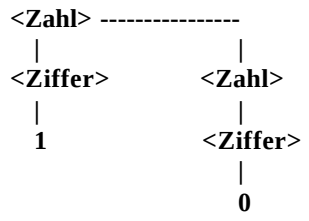
- **Ableitungssequenz**
- **Ableitungsbaum**

Eine Ableitungssequenz wird in der Form beschrieben, dass ausgehend vom Startsymbol jeweils ein Nonterminal durch eine Folge von Nonterminalen und Terminalen ersetzt wird, bis man beim gesuchten Wort ist.

Bsp.: (Zahl 10)
<Zahl> -> <Ziffer><Zahl> -> 1<Zahl> -> 1<Ziffer> -> 10
=> 10 ist eine Zahl

In einem Ableitungsbaum wird nicht nur jeweils ein Nonterminal durch eine Folge von Nonterminalen und Terminalen ersetzt, sondern sämtliche Nonterminalsymbole auf einer Ebene des Baums.

Ableitungsbaum für das Beispiel:



Ein Wort heißt eindeutig, wenn es nur einen Ableitungsbaum für das Wort gibt.

Ansonsten heißt es mehrdeutig.

Eine Grammatik, die mehrdeutige Wörter enthält, heißt mehrdeutig.

Ansonsten heißt sie eindeutig.

Beispiel:

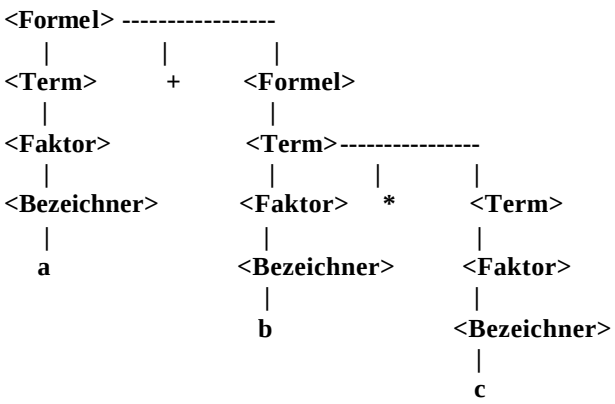
Zwei Grammatiken für arithmetische Ausdrücke:

<Formel>::=<Term> | <Term> + <Formel>
<Term>::=<Faktor> | <Faktor> * <Term>
<Faktor>::=<Bezeichner>

<Formel>::=<Formel> + <Formel> | <Formel> * <Formel> | <Bezeichner>

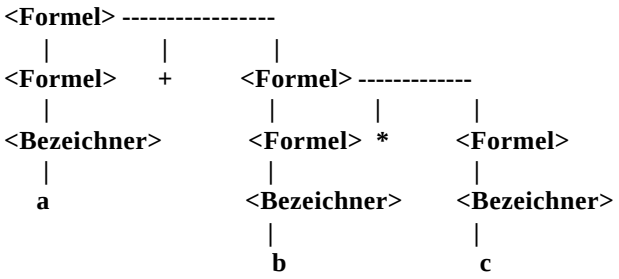
Die erste Grammatik ist eindeutig, die zweite ist mehrdeutig, wie anhand des Beispiels a+b*c zu sehen ist.

*Eindeutigkeit von a+b*c in der ersten Grammatik:*

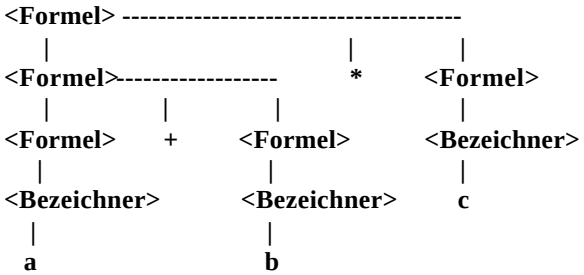


*Mehrdeutigkeit von a+b*c in der zweiten Grammatik:*

Variante 1:



Variante 2:



4. Aussagen- und Prädikatenlogik

4.1 Aussagenlogik

Aussage: Satz, von dem man entscheiden kann, ob er wahr oder falsch ist.

Aussagen werden auf die Wahrheitswerte w und f reduziert.

Bemerkung: Es gibt auch Fälle, in denen zusätzlich der Wahrheitswert unbekannt in die Betrachtung einbezogen werden muß!

(NULL Wert im relationalen Datenmodell)

Paradoxe werden nicht detailliert behandelt!

Operatoren der Aussagenlogik (Boole’sche Algebra)

Nicht

Und

Oder

Implikation

Äquivalenz

Definition durch Wahrheitstabellen

Normalformen:

Disjunktive Normalform

Konjunktive Normalform

Beweisführung für Wahrheitsgehalt aussagelogischer Formeln durch Wahrheitstabellen.

Wichtige aussagelogische Formeln:

Kommutativgesetze	**E1 und E2 = E2 und E1** **E1 oder E2 = E2 oder E1** **E1 ⇔ E2 = E2 ⇔ E1**
Assoziativgesetze	**E1 und (E2 und E3) = (E1 und E2) und E3** **E1 oder (E2 oder E3) = (E1 oder E2) oder E3**
Distributivgesetze	**E1 oder (E2 und E3) = (E1 oder E2) und (E1 oder E3)** **E1 und (E2 oder E3) = (E1 und E2) oder (E1 und E3)**
Regeln von De Morgan	**Nicht (E1 und E2) = (nicht E1) oder (nicht E2)** **Nicht (E1 oder E2) = (nicht E1) und (nicht E2)**
Doppelte Negation	**Nicht (nicht E1) = E1**
Gesetz vom Ausgeschlossenen Dritten	**(Nicht E1) oder E1 = T**
Gesetz vom Widerspruch	**(Nicht E1) und E1 = F**
Implikation	**E1 => E2 = (nicht E1) oder E2** **T=>E2 = E2**
Äquivalenz	**E1 ⇔ E2 = (E1 => E2) und (E2 => E1)**
Oder-Vereinfachung	**E1 oder E1 = E1** **E1 oder T = T** **E1 oder F = E1** **E1 oder (E1 und E2) = E1**
Und-Vereinfachung	**E1 und E1 = E1** **E1 und T = E1** **E1 und F = F** **E1 und (E1 oder E2) = E1**
Identität	**E1 = E1**

Formeln, die immer wahr sind, heißen Tautologie.

Formeln, die immer falsch sind, heißen Kontradiktion.

4.2 Prädikatenlogik

Prädikat: Aussageform, die von Variablen abhängt und für feste Variablenwerte eine Aussage repräsentiert.

Prädikate werden quantifiziert durch die Quantoren „Für alle“ und „Es existiert“

Beispiel:

Prädikat: P(x,y) = x-y >0, x,y natürliche Zahlen

Aussagen:

- **P(x,y), falls für x und y natürliche Zahlen eingesetzt werden**
- **Für alle natürlichen Zahlen x: Für alle natürlichen Zahlen y: P(x,y) (falsch)**
- **Es existiert eine natürliche Zahl x: Für alle natürlichen Zahlen y: P(x,y) (falsch)**
- **Für alle natürlichen Zahlen x: Es existiert eine natürliche Zahl y: P(x,y) (wahr)**
- **Es existiert eine natürliche Zahl x: Es existiert eine natürliche Zahl y: P(x,y) (wahr)**

Wichtige prädikatenlogische Formeln:

Nicht (Für alle x: A(x)) ⇔ Es existiert x: nicht A(x)

Nicht (Es existiert x: A(x)) ⇔ Für alle x: nicht A(x)

Es existiert x: A(x) ⇔ Nicht (Für alle x: nicht A(x)))

Für alle x: A(x) ⇔ Nicht (Es existiert x: nicht(A(x)))

Beweisführung:

Beweis, dass eine Aussage wahr ist:

Direkte Beweis:	**A=>B**
Indirekter Beweis:	**nicht B => nicht A**
Beweis durch Widerspruch:	**A und (nicht B) = F**

Beweis, dass eine Aussage der Form „Es existiert x: A(x)“ wahr ist:

konstruktiv

Beweis, dass eine Aussage der Form „Für alle x: A(x)“ falsch ist:

Gegenbeispiel

Beweis durch vollständige Induktion

Das Beweisprinzip der vollständigen Induktion basiert auf den Peano Axiomen für die natürlichen Zahlen. Es besagt, dass ein Prädikat für alle natürlichen Zahlen gilt, wenn man zeigen kann:

- **P(1) ist wahr (Induktionsbeginn)**
- **P(n) ist wahr für eine natürliche Zahl n => P(n+1) ist wahr (Induktionsschritt)**

Beispiel:

Die Summe der ersten n natürlichen Zahlen ist n*(n+1)/2:

Induktionsbeginn (n=1): Summe(i:i=1,...,1) = 1 = 1*2/2

Induktionsschritt (n=>n+1):

Falls Summe(i:i=1,...,n)=n*(n+1)/2, so folgt:
Summe(i:i=1,...,n+1)=n*(n+1)/2 + n+1 = (n+1)*(n/2+1) = (n+1)*(n+2)/2

Definition durch Induktion

Definition einer Folge von Objekten D(n) zu jeder natürlichen Zahl n durch

- **Definition von D(1)**
- **Definition von D(n+1) mit Hilfe von D(1),...,D(n) für n>1**

Derartige Definitionen werden als rekursiv bezeichnet. (Ein Begriff wird durch sich selbst definiert.)

Beispiel:

Fak(n) (n!) ist definiert durch

- **Fak(1)=1**
- **Fak(n+1)=(n+1)*Fak(n), n>1**

Exkurs in die Kombinatorik

Wie viele Möglichkeiten gibt es, aus n Objekten k Objekte auszuwählen, wobei

i. **Mehrfachauswahl erlaubt ist und alle Permutationen gezählt werden**

ii. **Mehrfachauswahl nicht erlaubt ist und alle Permutationen gezählt werden**

iii. **Mehrfachauswahl nicht erlaubt ist und jeweils nur 1 Permutation gezählt wird**

iv. **Mehrfachauswahl erlaubt ist und jeweils nur 1 Permutation gezählt wird**

i) **n^k**

ii) **n*(n-1)*...*(n-k+1) = fak(n) / fak(n-k) =:P(n,k)**

iii) **fak(n) / (fak(n-k)*fak(k)) =:C(n.k) (Binomialkoeffizient n über k)**

Es gelten folgende Aussagen für die Binomialkoeffizienten:

Für alle natürlichen Zahlen n und k mit 0<=k<=n gilt
C(n+1,k+1)=C(n,k) + C(n,k+1)

Für alle natürlichen Zahlen n und k gilt:
C(n+k,k)=1+C(n,1)+C(n+1,2)+...+C(n+k-1,k)

iv) **Für S(n,k) gilt die Rekursionsgleichung:**

S(1,k)=1 für alle k>=1
S(n+1,k)=1+S(n,1)+...+S(n,k) für natürliche Zahlen n und alle k>=1

Beweis:
Um k Elemente aus n+1 Elementen auszuwählen, kann man
das n+1-te Element k-fach auswählen,
1 Elemente aus n Elementen auswählen und das n+1-te Element k-1-fach,
2 Elemente aus n Elementen auswählen und das n+1-te Element k-2-fach
...
k Elemente aus n Elementen auswählen

Aus dieser Rekursionsgleichung folgt S(n,k) = C(n+k-1,k) für alle natürlichen Zahlen n und k, wie durch vollständige Induktion bewiesen wird.

5. **Korrektheit von Algorithmen**

Verwendung von Assertions:

- **Pre Condition (Vorbedingung)**
- **Post Condition (Nachbedingung)**

Vorbedingung: Aussage, die vor Ausführung des Algorithmus gilt.

Nachbedingung: Aussage, die nach Ausführung des Algorithmus gelten soll.

Korrektheitsbeweis:

Falls die Vorbedingung P gilt, so gilt nach Ausführung des Algorithmus A die Nachbedingung Q: {P}A{Q}

Für den Beweis der Korrektheit eines Algorithmus sind folgende Aussagen zur Korrektheit von Sequenz, Selektion und Iteration wichtig:

Gilt {P}A1{Q1} und {Q1}A2{Q}, so gilt: {P}A1;A2{Q}

Gilt {P und Bedi ngung}A{Q} und (P und nicht Bedingung) => Q, so gilt {P}Falls Bedingung, dann A{Q}
Gilt {P und Bedingung}A1{Q} und {P und nicht Bedingung}A2{Q}, so gilt {P}Falls Bedingung, dann A1, sonst A2{Q}

Für die Betrachtung der Iteration wird der Begriff der Schleifeninvarianten SI eingeführt. Dies ist eine Aussage, die vor der Iteration der Vorbedingung entspricht, nach der Iteration der Nachbedingung und nach jedem Schleifendurchlauf wahr ist.

Aus {SI}S{SI} folgt dann {P}Solange Bedingung, S{Q}

Beispiel:

Berechnung der Summe der ersten n natürlichen Zahlen.

1. Setze sum auf 0.
{sum=0}
2. Setze i auf 1.
{sum=0 und i=1}
3. Solange i kleiner oder gleich n ist, führe Schritte 4 und 5 durch:
{sum=Summe der natürlichen Zahlen von 1 bis i-1, i<=n}
4.Setze sum auf sum+i
{sum=Summe der natürlichen Zahlen von 1 bis i, i<=n}
5.Setze i auf i+1
{sum=Summe der natürlichen Zahlen von 1 bis i-1, i<=n+1}
{sum=Summe der natürlichen Zahlen von 1 bis n}

Problem der Korrektheitsbeweise von Algorithmen:

Beweise sind bereits für einfache Algorithmen aufwendig. Daher werden sie häufig durch Plausibilitätsbetrachtungen ersetzt.

6. Mengen, Relationen und Funktionen

6.1 Mengen

Eine Menge wird definiert als Zusammenfassung der Elemente, die zu ihr gehören.

Mengen können definiert werden durch

- **Enumeration (wobei Duplikate nicht erlaubt sind)**
- **Prädikate (mit Bezug auf eine Basismenge)**

Eine spezielle Menge ist die Menge ohne Elemente, die leere Menge {} bzw. 0.

Beispiel:

Die Menge der ersten 3 natürlichen Zahlen kann definiert werden als

{1,2,3}

oder

{n: n ist natürliche Zahl und n<=3}

Im Allgemeinen wird die zweite Form der Definition verwendet. Dies bedingt, dass man geeignete Basismengen zur Verfügung hat. Wir verwenden als Basismengen die natürlichen, die ganzen, die rationalen, die reellen und die komplexen Zahlen.

Operationen auf Mengen

Reduktion auf Operationen der Aussagenlogik

Teilmenge, Obermenge

Durchschnitt, Vereinigung

Durchschnitt und Vereinigung von n>2 Mengen werden rekursiv definiert.

Differenz, Symmetrische Differenz

Daher ergeben sich für Mengen analoge Formeln zu denen der Aussagelogik (Übung).

Wichtige mengentheoretische Begriffe:

Potenzmenge P(M) einer Menge M: Menge der Teilmengen von M.

Kartesisches Produkt zweier Mengen M1 und M2: M1xM2={(m1,m2):m1 in M1 und m2 in M2}. Das kartesische Produkt von n Mengen wird per Induktion definiert.

Partition einer Menge M: Menge von nichtleeren Teilmengen M1,..., Mn von M, deren Vereinigung M ist (Überdeckung von M) und die paarweise disjunkt sind.

Kardinalität #M einer (endlichen) Menge M: Anzahl der Elemente

Die Kardinalität der Potenzmenge einer Menge M mit #M=n ist 2^n.

Die Kardinalität des kartesischen Produkts von Mengen ist das Produkt der Kardinalitäten.

Ist {M1,...,Mn} eine Partition von M, so ist die Kardinalität von M die Summe der Kardinalitäten von M1,...,Mn.

#(Vereinigung von M1 und M2) = #M1 + #M2 - #(Durchschnitt von M1 und M2)

6.2 Relationen

**Der Begriff „Relation“ kann durch „Beziehung“ übersetzt werden.
Mathematisch wird dies wie folgt präzisiert:**

Seien M1 und M2 Mengen. Eine binäre Relation R zwischen Elementen von M1 und M2 ist eine Teilmenge des kartesischen Produkts M1xM2.

Ist (m1,m2) in R (also x steht in Beziehung zu y), so schreiben wir auch: m1Rm2.

Für den Fall M1=M2=:M sprechen wir von einer Relation auf M.

Neben binären Relationen kann man Relationen höheren Grades definieren, dies werden wir jedoch zunächst nicht weiterverfolgen.

Repräsentation von Relationen:

- **Pfeildiagramm**
- **Gerichteter Graph**
- **Matrix**

Eigenschaften von Relationen:

Eine Relation R auf einer Menge M heißt

- **reflexiv, wenn für alle x in M gilt: xRx**
- **symmetrisch, wenn gilt: xRy => yRx**
- **antisymmetrisch, wenn gilt: xRy and yRx => x=y**
- **asymmetrisch, wenn gilt: xRy => nicht (yRx)**
- **transitiv, wenn gilt: xRy and yRz => xRz**

Eine Relation heißt partielle Ordnungsrelation (Halbordnung), wenn sie reflexiv, antisymmetrisch und transitiv ist.

Eine Relation R heißt vollständige Ordnungsrelation (Ordnung) auf einer Menge M, wenn sie asymmetrisch und transitiv ist und für alle x,y in M eine der folgenden Aussagen wahr ist: xRy, yRx, x=y.

Eine vollständige Ordnungsrelation wird auch als lineare Ordnung bezeichnet.

Ist R(i) eine lineare Ordnung auf M(i), so kann durch

(x1,...,xn)Rn(y1,...,yn) :⇔

Für das minimale i in {1,...,n} mit x(i) != y(i) gilt x(i) R(i) y(i)

eine lexikographische Ordnung auf M^n definiert werden.

Beispiel:

Lexikographische Ordnung auf Strings, die auf der vollständigen Ordnung des zugrundeliegenden Zeichensatzes beruht.

Äquivalenzrelationen

Eine Relation heißt Äquivalenzrelation, wenn sie reflexiv, symmetrisch und transitiv ist.

Jede Äquivalenzrelation erzeugt eine Partition einer Menge vice versa dadurch, dass folgende Definition (in der jeweils interessierenden Richtung) angewandt wird:

X ist äquivalent zu y genau dann, wenn x und y zur gleichen Teilmenge der Partition gehören.

Transitive Hülle:

Minimale Relation, die transitiv ist und Obermenge einer gegebenen Relation ist.

Bedeutung der transitiven Hülle:

Erreichbarkeit

Berechnung der transitiven Hülle:

Graphentheorie (Warshall)

Matrixmultiplikation:

Sei M die Matrix, die die Relation beschreibt. Wenn M^n berechnet wird und M^n = M^(n-1) gilt, dann ist M^n die Matrixdarstellung der transitiven Hülle von M. Als Approximation für die transitive Hülle wird häufig M^N für einen kleinen Wert N gewählt.

Definitionen:

Die Relation id auf M: id={(m,m):m in M} wird als Identität auf M bezeichnet.

Sei R eine Relation auf M. Dann wird R^-1, die zu R inverse Relation, definiert durch: R^-1:={(m2,m1) in MxM: (m1,m2) in R}

Seien R1 eine Relation in M1xM2 und R2 eine Relation in M2xM3. Dann wird die Verknüpfung R2*R1 definiert durch

R2*R1:={(m1,m3) in M1xM3: Es existiert m2 in M2: (m1,m2) in R1 und (m2,m3) in R2)}

Aussagen:

- **Für die Verknüpfung von Relationen gilt das Assoziativgesetz.**
- **Eine Relation auf M ist reflexiv genau dann, wenn sie Obermenge der Identität ist.**
- **Eine Relation auf M ist symmetrisch genau dann, wenn glt: R=R^-1.**
- **Eine Relation auf M ist transitiv genau dann, wenn R*R Teilmenge von R ist.**

6.3 Funktionen:

Unter einer Funktion versteht man intuitiv eine Abbildungsvorschrift, die einer gewissen Menge von Werten x (dem Definitionsbereich der Funktion) einen Funktionswert f(x) zuordnet.

Wir wollen hier eine etwas abstraktere Sicht einnehmen:

Eine Funktion von M1 nach M2 ist eine Relation R in M1xM2 mit der Eigenschaft, dass aus (m,n1) in R und (m,n2) in R n1=n2 folgt.

{m in M1: Es existiert ein n in M2 mit (m,n) in R} heißt Definitionsbereich von R,

{n in M2: Es existiert ein m in M1 mit (m,n) in R} heißt Wertebereich von R.

Funktionen werden i.a. in Kleinschreibung bezeichnet – im Gegensatz zu Relationen, die i.a. in Großschreibung bezeichnet werden.

Für Funktionen werden folgende Begriffe definiert:

- **Eine Funktion heißt injektiv, wenn aus f(x1)=f(x2) folgt: x1=x2.**
- **Eine Funktion f:X->Y heißt surjektiv, wenn zu jedem y in Y ein x in X existiert mit y=f(x).**
- **Eine Funktion heißt bijektiv, wenn sie injektiv und surjektiv ist.**

Der Begriff der Bijektion ist wird herangezogen, wenn es um den Begriff der Abzählbarkeit von Mengen geht:

- **Eine Menge M heißt abzählbar, wenn es eine bijektive Funktion von M in die natürlichen Zahlen gibt.**
- **Eine Menge, die nicht abzählbar ist, heißt überabzählbar.**
- **Es gibt überabzählbare Mengen, z.B. die Menge der Funktionen von den natürlichen Zahlen in die natürlichen Zahlen.**

7. Berechenbarkeit und Entscheidbarkeit

Die Begriffe der Menge und der Funktion spielen eine wesentliche Definition bei der formalen Definition des Begriffs Berechenbarkeit:

Eine Funktion f wird als berechenbar bezeichnet, wenn es einen Algorithmus gibt, der f(x) für alle x liefert.

Ist f eine partielle Funktion, also nicht überall definiert, so wird f als berechenbar bezeichnet, wenn es einen Algorithmus gibt, der f(x) für alle x aus dem Definitionsbereich von f liefert und ansonsten nicht terminiert.

Eine Menge heißt entscheidbar, wenn die charakteristische Funktion dieser Menge berechenbar ist.

Eine Menge ist also entscheidbar, wenn es einen Algorithmus gibt, der für alle x das Resultat liefert, ob x zu der Menge gehört oder nicht. Der Entscheidbarkeitsbegriff für Mengen entspricht dem Begriff der Lösbarkeit von Problemen, wenn man die Menge aller Eingabegrößen, für die ein gewisses Problem lösbar ist, zugrundelegt.

Beispiele:

1. Die Funktion f mit leerem Definitionsbereich ist berechenbar.

2. Die Funktion auf den natürlichen Zahlen, die durch

- **f(n)=1, falls n ein Anfangsabschnitt aus der Dezimalbruchentwicklung von PI ist,**
- **f(n)=0 sonst**

definiert ist, ist berechenbar.

3. Die Funktion auf den natürlichen Zahlen, die durch

- **f(n)=1, falls n in der Dezimalbruchentwicklung von PI vorkommt,**
- **f(n)=0 sonst**

definiert ist, ist nicht berechenbar.

Es stellt sich die Frage, warum ein derart formaler Begriff der Berechenbarkeit und damit der Entscheidbarkeit eingeführt wird. Um zu zeigen, dass ein Problem lösbar ist, genügt es doch, einen Algorithmus zu finden, der das Problem löst!

Auf der anderen Seite gibt es aber unlösbare Probleme (und zwar mehr als lösbare Probleme!), und um zu zeigen, dass ein Problem nicht lösbar ist, bedarf es eines formalen Begriffs der Entscheidbarkeit und damit der Berechenbarkeit. Dieser Begriff sollte natürlich mit dem intuitiven Berechenbarkeitsbegriff übereinstimmen.

Die Church'sche These sagt aus, dass verschiedene formale Berechenbarkeitsbegriffe äquivalent sind und mit dem intuitiven Berechenbarkeitsbegriff übereinstimmen. Der erste Teil der Aussage ist beweisbar, der zweite kann es nicht sein, weil der Begriff der intuitiven Berechenbarkeit nicht formalisiert ist!

Es gibt einige Modelle zur Berechenbarkeit:

- **Turing Maschine**
- **While Programm**
- **Goto Programm**
- **RAM (Register Address Machine)**

Wir werden uns kurz auf die Turing Maschine und das While Programm eingehen. Die Turing Maschine hat den Vorteil, dass sie für Beweise, dass Probleme unlösbar sind, sehr gut herangezogen werden kann, das While Programm ist „ziemlich intuitiv".

Turing Maschine:

Die Vorstellung, die man mit einer Turing Maschine verbinden kann, ist die, dass man ein (unendliches) Band hat, auf dem zunächst eine Menge von Eingabezeichen steht, ein Eingabewort. Der Schreib-Lesekopf befindet sich auf dem ersten Zeichen des Eingabeworts. Dies wird nun verarbeitet, wobei die Turing Maschine zustandsorientiert ist und aus gegebenem Zustand und gelesenem Zeichen den Folgezustand bestimmt; zusätzlich wird das Ausgabezeichen bestimmt und die Richtung, in der das Band weiter gelesen wird. Wird ein Zustand erreicht, der in einer vorgegebenen Menge von Endzuständen ist, so endet die Verarbeitung. Häufig wird gefordert, dass der Schreib-Lesekopf auf dem ersten Zeichen des Ausgabeworts steht.

Formal kann man die Definition wie folgt fassen:

Eine (deterministische) Turing Maschine ist ein 7-Tupel, bestehend aus

- **Zustandsmenge**
- **Eingabealphabet**
- **Ausgabealphabet (Obermenge des Eingabealphabets)**
- **Überführungsfunktion, die aus gegebenem Zustand und Eingabezeichen den Folgezustand, das Ausgabezeichen und die Leserichtung bestimmt**
- **Anfangszustand**
- **Spezielles Zeichen „Blank“**
- **Menge der Endzustände**

Beispiel:

Eine Turing Maschine, die an eine Folge von Zeichen A ein weiteres Zeichen A anhängt.

Band vor der Verarbeitung:

A	**A**	**A**	**A**	**Blank**

Eingabealphabet: {Blank, A}

Ausgabealphabet: {Blank,A}

Zustandsmenge:

- **Z0: Schreib-Lesekopf steht auf 1. Zeichen des Eingabeworts**
- **Z1: Verarbeitung im Gange**
- **Z2: Blank erreicht**
- **Z3: Schreib-Lesekopf steht auf 1. Zeichen des Ausgabeworts**

Übergangsfunktion:

	A	**Blank**
Z0	**Z1,A,R**	**Z2,A,L**
Z1	**Z1,A,R**	**Z2,A,L**
Z2	**Z2,A,L**	**Z3,Blank,R**

Dualer Addierer

Eingabealphabet: {0,1,Blank}

Ausgabealphabet: {0,1,Blank}

Zustandsmenge:

- **Z0: Schreib-Lesekopf steht auf 1. Zeichen des Eingabeworts**
- **Z1: Lesen des Bands von links nach rechts**
- **Z2: Lesen des Bands von rechts nach links (nach Addition der 1)**
- **Z3: Lesen des Bands von rechts nach links, nachdem die erste 0 erreicht wurde**

Z4: Schreib-Lesekopf steht auf dem 1. Zeichen des Ausgabeworts

Übergangsfunktion:

	0	**1**	**Blank**
Z0	**Z1,0,R**	**Z1,1,R**	**Z4,1,N**
Z1	**Z1,0,R**	**Z1,1,R**	**Z2,Blank,L**
Z2	**Z3,1,L**	**Z2,0,L**	**Z4,1,N**
Z3	**Z3,0,L**	**Z3,1,L**	**Z4,Blank,R**

While Programme:

While Programme bestehen aus folgenden syntaktischen Komponenten:

- **Variablen x0 x1 x2 ...**
- **Konstanten 0 1 2 ...**
- **Operationssymbole + -**
- **Trennsymbole ; := = !=**
- **Schlüsselwörter WHILE DO END LOOP**

Jede Wertzuweisung der Form

x:= x+c und x:=x-c (x Variable, c Konstante)

ist ein While Programm.

Sind P1 und P2 While Programme, so ist es auch die Sequenz P1;P2.

Ist P1 ein While Programm, so auch das Konstrukt

WHILE x(i) != 0 DO P1 END

Ist P1 ein While Programm, so auch das Konstrukt

LOOP x DO P1 END

While Programme beinhalten die Ablaufstrukturen Sequenz und Iteration, kommen also dem, was wir als Algorithmus bezeichnet haben, ziemlich nahe. Es scheint jedoch die Selektion als Ablaufstruktur zu fehlen. Dies ist allerdings nicht der Fall, denn die Selektion

IF x1 THEN P1 ELSE P2

kann wie folgt durch ein While Programm simuliert werden:

x2:=0

x3:=1

LOOP x1 DO x2:=1; x3:=0 END;

LOOP x2 DO P1 END;

LOOP x3 DO P2 END

Es gilt nun folgende Aussage:

Eine Funktion ist genau dann While berechenbar, wenn sie Turing berechenbar ist.

Das Halteproblem:

Gibt es, einen Algorithmus, der für einen gegebenen Algorithmus a priori bestimmen kann, ob er korrekt ist oder nicht.

Es kann gezeigt werden, dass das Halteproblem nicht lösbar ist.

Obwohl die Behandlung des Halteproblems eine ziemlich theoretische Angelegenheit ist, ist diese Konsequenz für die praktische Informatik sehr wichtig (Testkonzept)

8. Komplexität

Entscheidbarkeit beschäftigt sich mit der Frage: Ist ein Problem überhaupt lösbar?

Komplexität ist ein Maß, wie effizient ein Problem lösbar ist.

Komplexität wird definiert für

- **Algorithmen**
- **Probleme**

Die Komplexität eines Problems wird zurückgeführt auf die Komplexität von Algorithmen:

Komplexität eines Problems: minimale Komplexität eines Algorithmus, der das Problem löst.

Wir betrachten im folgenden die Komplexität von Algorithmen.

Die Komplexität eines Algorithmus ist orientiert an der Anzahl der elementaren Schritte, die bei der Ausführung des Algorithmus anfallen, in Abhängigkeit von der Kardinalität der Eingabe.

Da dieser Wert für verschiedene Eingaben gleicher Kardinalität i.a. unterschiedlich ist, werden i.a. 3 Betrachtungen zur Komplexität eines Algorithmus angestellt:

- **Best case analysis**
- **Average case analysis**
- **Worst case analysis**

Im Rahmen der Komplexitätsanalyse beschäftigt man sich nicht mit exakten Werten, sondern mit dem Verhalten des Algorithmus für große Eingabemengen. Daraus wird abgeleitet, ob ein Algorithmus auch für große Eingabemengen zur effizienten Lösung eines Problems geeignet ist.

Zur Angabe der Komplexität werden häufig die Landau'schen Symbole werwendet:

f(x) = O(g(x)) heißt dabei, dass eine Konstante C>0 existiert mit f(x) <=C*g(x) für alle x.

f(x) = 0(g(x)) heißt dabei, dass eine Konstante C>0 existiert mit f(x) >=C*g(x) für alle x.

Berechnung der Komplexität eines Algorithmus:

Operation	**Anzahl Rechenschritte**
Arithmetik, Vergleich, Wertzuweisung	**1**
Ein- und Ausgabe	**1**
Funktionsaufruf	**Komplexität der Funktion**
Sequenz	**Summe der Einzelschritte**
Selektion	**Komplexität des logischen Ausdrucks +** **Maximum der Alternativen**
Iteration (m Durchläufe)	**Initialisierung +** **M*Komplexität des logischen Ausdrucks + M*Komplexität des Schleifenkörpers +** **M*Komplexität des Weiterzählens**

Beispiel:

Berechnung der Summe der ersten n natürlichen Zahlen:

1*Eingabe

1*Wertzuweisung (Summe)

1*Wertzuweisung (Initialisierung der Schleife)

n*Vergleich (Komplexität des logischen Ausdrucks in der Schleife)

n*Wertzuweisung (Komplexität des Schleifenkörpers)

n*Wertzuweisung (Komplexität des Weiterzählens in der Schleife)

1*Ausgabe

Dies ergibt eine Komplexität von 3*n+4 =O(n). Der Algorithmus weist lineare Komplexität auf.

Mit dem Thema Komplexität von Algorithmen werden wir uns im Kapitel „Lösungsverfahren“ und im zweiten Semester noch ausführlich beschäftigen.

Wir werden dann auch mit einem zweiten Komplexitätsmaß betrachten, welchen Speicherplatz ein Algorithmus benötigt. Wir unterscheiden dann

- **Zeitkomplexität**

- **Raumkomplexität**

Der Schwerpunkt der Betrachtung liegt jeweils auf der Zeitkomplexität.