

---

# Theoretische Informatik

---

Sommersemester 2022  
INF2/ICS2

Prof. Dr. Georg Schied  
(Georg.Schied@thu.de)  
Fakultät Informatik

---

# Lektion 1

---

In dieser Lektion erfahren Sie zunächst, wie die Veranstaltung in diesem Semester organisiert ist. Danach bekommen Sie einen ersten Einblick in die Theoretische Informatik - wie sie als Themengebiet insgesamt in die Informatik einzuordnen ist, welche Teilbereiche dazu gehören und auch exemplarisch, in welcher Weise sie die Basis für Anwendungen bilden kann.

## Organisation

---

### *Inverted-Classroom-Konzept*

Die Online-Form dieser Veranstaltung wird im Wesentlichen nach dem Prinzip des **Inverted Classroom**, d.h. mit Schwerpunkt auf **Selbststudium**, organisiert sein.

### *Wöchentlicher Ablauf*

- ▶ **Zweimal pro Woche** (Mittwoch und Donnerstag) stelle ich Ihnen jeweils eine schriftliche **Lektion** via Moodle zur Verfügung (ca. 14-16 Seiten), die Sie innerhalb einer Woche durcharbeiten sollen.
- ▶ Zu jeder Lektion gibt es ein obligatorisches **Moodle-Quiz**.
  - Sie haben jeweils **eine Woche Zeit**, die Quizfragen zu beantworten.
  - Das erfolgreiche Bearbeiten dieser Quizfragen ist Teil der **Studienleistung** (Schein)!
  - Jeder Quiz kann beliebig oft wiederholt werden, so dass es kein Problem sein sollte, den Quiz erfolgreich abzuschließen.
- ▶ Es gibt **pro Woche ein Aufgabenblatt** mit obligatorischen "Scheinaufgaben" für die Studienleistung sowie mit weiteren nicht-obligatorischen Aufgaben (mehr dazu s.u.).
- ▶ Es gibt zwei **Vorlesungstermine** gemäß Stundenplan, im Normalfall mit folgender inhaltlicher Ausrichtung:
  - **Mittwoch 9.50-11:20 Uhr**: Wiederholung und Zusammenfassung der Lektionen der vorigen Wochen, Besprechung der in den Lektionen enthaltenen Aufgaben.
  - **Donnerstag 11:30-13:00 Uhr**: Besprechung von Aufgaben der Aufgabenblätter, weitere Beispiele und Übungen

- ▶ Sie können während der Woche jederzeit **Fragen stellen** über **Moodle** oder per **E-Mail** ([Georg.Schied@thu.de](mailto:Georg.Schied@thu.de)) :
  - im Betreff bitte TINF mit angeben
  - Wenn Sie unsicher sind, welche Anrede Sie verwenden sollen: z.B. "Hallo Herr Schied, ... "

## Aufgabenblätter

---

### *Scheinaufgaben*

- ▶ Die Abgabe der Lösungen erfolgt über Moodle. Der **Abgabetermin** für die Scheinaufgaben ist jeweils in der **folgenden Woche**. Beachten Sie den auf dem Aufgabenblatt angegebenen Abgabetermin, eine verspätete Abgabe ist nicht möglich!
- ▶ Die Lösungen für die Scheinaufgaben sind als **.pdf-Datei** abzugeben (kann auch handschriftlich erstellt, eingescannt/abfotografiert und in pdf gewandelt sein – gute Lesbarkeit vorausgesetzt).
- ▶ Es sind **2er-Teams** für die Bearbeitung und Abgabe der Scheinaufgaben **erlaubt**.
  - Sie müssen einer Moodle-Gruppe angehören, um Lösungen hochladen zu können.
  - Sprechen Sie sich mit dem Partner ab, wenn Sie eine 2er-Gruppen bilden wollen. Tragen Sie sich nicht ohne Absprache in eine andere Gruppe ein!
  - Wenn Sie einzeln abgeben wollen, bilden Sie eine 1er-Gruppe.
  - Pro Gruppe muss nur einer die Lösung hochladen, sie ist dann für beide Gruppenmitglieder sichtbar.
- ▶ Ihre Lösungen werden korrigiert. Sie erhalten über Moodle dann ein (kurzes) Feedback.
- ▶ Jedes Aufgabenblatt erhält eine Bewertung **bestanden/nicht bestanden**
  - zum Bestehen müssen jeweils 50% der Punkte erreicht werden
  - für den Schein müssen  $n-1$  der  $n$  Aufgabenblätter mit Scheinaufgaben bestanden sein (voraussichtlich  $n = 11$  in diesem Semester).
- ▶ Nach der Korrektur erhalten Sie die Musterlösungen für die Aufgaben

**Wichtig:** Abgeschriebene/kopierte Lösungen werden nicht akzeptiert und führen dazu, dass das Aufgabenblatt als nicht bestanden bewertet wird!

- ▶ Es ist aber durchaus erlaubt (und kann auch fürs Lernen sehr hilfreich sein), dass Sie sich gegenseitig in angemessener Weise kleine Tipps geben, wie etwas zu verstehen ist oder wie man selbst etwas gelöst hat – ohne aber komplette Lösungen weiterzugeben.
- ▶ Sie sollten niemals Lösungen für Aufgaben irgendwo für andere sichtbar zur Verfügung stellen (soziale Medien, Foren, ...). Das ist vielleicht gut gemeint als Unterstützung für Kommilitonen, die sich schwerer tun – aber "gut gemeint" ist das Gegenteil von "gut". Sie verleiten andere Leute damit nur dazu, sich selbst weniger mit dem Stoff zu beschäftigen, so dass diese letztendlich am Ende in der Klausur umso mehr Probleme haben werden.

## Hinweise zum eigenständigen Arbeiten

---

### *Erwartungen und Empfehlungen*

- ❑ Sie sollten pro Woche ca. 3 - 5 Stunden für diese Veranstaltung aufwenden zusätzlich zu den Vorlesungsterminen.
- ❑ Machen Sie sich einen Wochenplan, in dem Sie feste Zeiten für die Bearbeitung der Lektionen und der Aufgabenblätter vorsehen.
- ❑ Ich empfehle sehr, kleine Lerngruppen aus ca. 2 - 4 Leuten zu bilden, um sich gegenseitig beim Erarbeiten des Stoffs zu unterstützen.

### *Eine kleine "Bedienungsanleitung" für die Lektionen*

Zum Durcharbeiten der Lektionen empfehle ich folgende Vorgehensweise:

#### **Phase 1 - Vorbereiten (ca. 3 Min.):**

- ▶ Sorgen Sie dafür, dass Sie einen geeigneten Arbeitsplatz haben, an dem Sie ungestört und konzentriert arbeiten können.
- ▶ Kurz nachschauen, was in der vorigen Lektion behandelt wurde, um den Anschluss zu finden.
- ▶ Die Lektion schnell durchblättern, um einen Überblick zu bekommen: Wie viele Seiten sind es? Welche Themen sind in den Überschriften zu erkennen?

**Phase 2 - Durcharbeiten (ca. 45 - 60 Min.):** Die Lektion sorgfältig von vorne nach hinten durchlesen.

- ▶ Erstellen Sie eigene Notizen und fassen Sie den Inhalt zusammen. Diese Notizen haben die Funktion einer Vorlesungsmitschrift und sind ein wichtiges Hilfsmittel später zum Lernen auf die Klausur.
- ▶ Nach ca. der Hälfte der Lektion ist eine kurze Pause empfehlenswert (neuen Kaffee holen :-))

- ▶ Notieren Sie sich die Punkte, die Sie noch nicht verstanden haben

### **Phase 3 - Nacharbeiten und üben (ca. 10 - 15 Min.):**

- ▶ Gehen Sie nochmal die Teile durch, die Sie nicht verstanden haben und lösen Sie die enthaltenen Aufgaben (falls nicht schon in Phase 2 gemacht).
- ▶ Notieren Sie sich alle verbleibenden Probleme, so dass Sie in der nächsten Veranstaltung danach fragen können.

### **Phase 4 - Abschluss (ca. 5 Min.):**

- ▶ Lösen Sie das Moodle-Quiz zur Lektion, Blättern Sie ggf. in der Lektion nach, falls Sie die Lösung für eine Quizfrage nicht gleich wissen.
- ▶ Loben Sie sich am Ende dafür, dass Sie so fleißig gearbeitet haben!

## Studien- und Prüfungsleistungen

---

### *Studienleistung ("Schein")*

- ❑  **$k - 3$  der  $k$  Quizze** für die Lektionen (voraussichtlich  $k = 25$ ) müssen erfolgreich bearbeitet sein (von jedem individuell)
- ❑  **$n - 1$  von  $n$  Aufgabenblättern** mit Scheinaufgaben (voraussichtlich  $n = 11$ ) müssen als bestanden bewertet worden sein (einzeln oder in 2er-Gruppe)

### *Prüfungsleistung*

- ▶ Voraussetzung: erfolgreiche Studienleistung s.o.
- ▶ Klausur 90 Min., ohne Hilfsmittel

# Kap. 1 Einführung und Motivation

## Inhalt

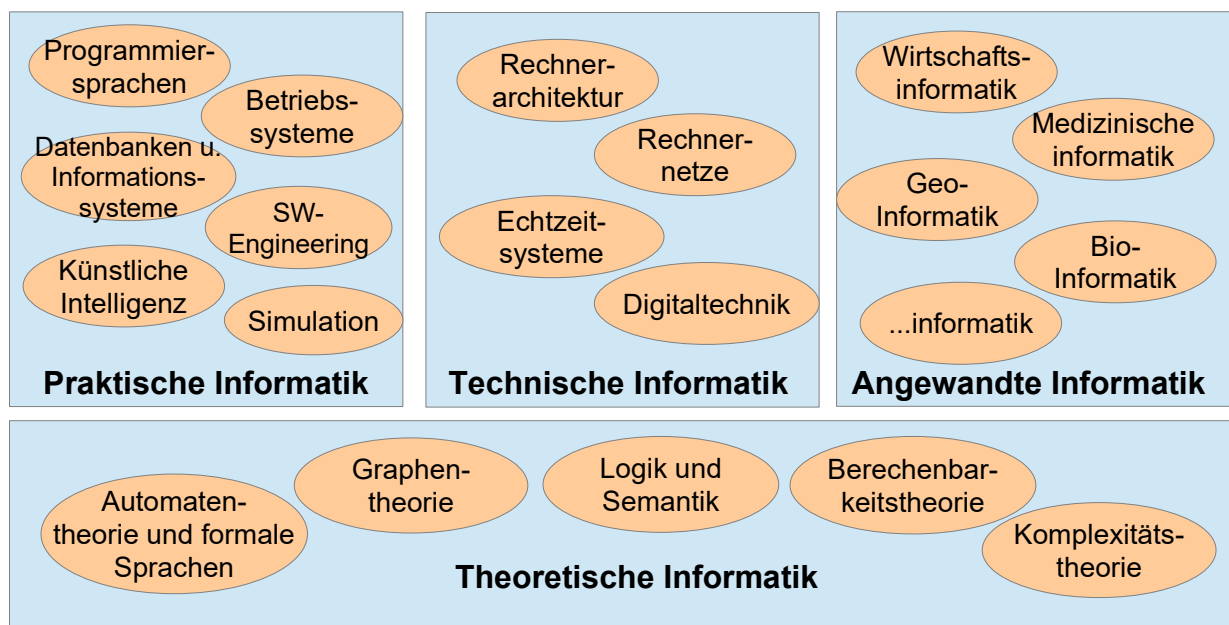
- ▶ Was ist Theoretische Informatik?
- ▶ Warum ist theoretische Informatik wichtig?
- ▶ Ziele und Inhalt der Lehrveranstaltung
- ▶ Literatur und Materialien

## 1.1 Was ist Theoretische Informatik?

Unter dem Begriff "Theoretische Informatik" werden verschiedene Themengebiete zusammengefasst, die, ähnlich wie die Mathematik, eine formale Grundlage für andere, anwendungsnähere Teilgebiete der Informatik bilden.

### Teilgebiete der Informatik

Eine gängige Übersicht über Teilgebiete der Informatik sehen Sie in der folgenden Abbildung. Die Theoretische Informatik ist als "Fundament" unter den anderen Themenbereichen dargestellt.



Die wichtigsten Teilgebiete der theoretischen Informatik sind folgende:

### Automatentheorie und formale Sprachen

- ▶ Wie definiert man exakt die Syntax einer Programmiersprache oder anderer formaler Beschreibungssprachen (z.B. Datenbeschreibungssprachen wie XML,

JSON oder Dokumentenbeschreibungssprachen wie HTML)? Und wie kann man systematisch und effizient Syntaxanalyse machen, d.h. prüfen, ob ein Programm/Dokument syntaktisch korrekt aufgebaut ist und es analysieren, so dass dann eine Weiterverarbeitung möglich ist?

- ▶ Dieses Teilgebiet ist die Grundlage für eines der Hauptwerkzeuge, die Sie als Informatiker verwenden: **Programmiersprachen** und deren Implementierung mittels Compilern oder Interpretern.

## Graphentheorie

- ▶ Graphen sind ein mathematisches, formales Beschreibungskonzept. Viele Problem in der Informatik werden zunächst mit Hilfe von Graphen auf abstrakter Ebene formalisiert und auf dieser abstrakten Beschreibungsebene werden dann Lösungen erarbeitet. Die Lösung am Ende auf programmiersprachlicher Ebene umzusetzen, ist dann eher "nur noch" Handwerk – das Sie als Informatiker/Informatikerin natürlich auch gut beherrschen müssen.
- ▶ Der Bereich der Graphentheorie liegt im Schnittbereich zwischen Theoretischer Informatik und (diskreter) Mathematik. Sie werden in diesem Semester nur einige wichtige Grundbegriffe kennenlernen – um auf dieser abstrakten Ebene "mitreden" zu können. Im nächsten Semester lernen Sie dann in der Vorlesung *Algorithmen und Datenstrukturen* interessante Algorithmen für graphbasierte Fragestellungen kennen.

## Logik und Semantik

Die formale Logik stammt ursprünglich aus dem Bereich der Grundlagen der Mathematik. In Ihrer Mathematik-Vorlesung im ersten Semester haben Sie schon die Aussagenlogik als ein Beispiel einer formalen Logik kennengelernt, und in Programmieren haben Sie diese Logik in Form von logischen Ausdrücken mit  $\&$ ,  $||$ ,  $!$  auch schon fleißig angewendet.

- ▶ Die Aussagenlogik ist allerdings sehr eingeschränkt in dem, was mit ihr ausgedrückt werden kann. Es gibt mächtigere, aussagekräftigere Logik-Systeme, die in der Mathematik und Informatik eine wichtige Rolle spielen. Die **Prädikatenlogik** werden Sie im Verlauf des Semester genauer kennenlernen.
- ▶ Unter **Semantik** versteht man die Bedeutung von Sprachen. Den Begriff haben Sie vermutlich auch schon bei der Einführung ins Programmieren kennengelernt. Bei formalen Logiksystemen ist Syntax und Semantik wichtig: Wie schreibe ich eine Aussage als Formel hin (das gibt die Syntax vor) und was bedeutet eine Formel dann (dazu hat eine Logik eine mathematisch exakt definierte Semantik).

## Berechenbarkeitstheorie

Es geht im wesentlichen um folgende zwei sehr prinzipielle Fragestellungen:

- ▶ Was kann mit einem Computer (d.h. einer "Rechenmaschine") überhaupt berechnet werden? Kann z.B. jede mathematische Funktion berechnet werden?
- ▶ Was ist an Grundbefehlen und Grundkonstrukten nötig, damit eine "Rechenmaschine" alles berechnen kann, was berechenbar ist?

Dieser Teil der theoretischen Informatik ist der, der wohl am weitesten von direkten Anwendungen entfernt ist. Trotzdem können gerade diese grundsätzlichen Fragestellungen faszinierend sein: Es gibt Dinge, die prinzipiell unmöglich sind – nicht nur, weil unsere Rechner jetzt zu lange dazu brauchen würden (auch wenn Sie 1 000 000 mal schneller wären) oder zu wenig Speicher haben.

Und letztendlich hat es in manchen Fällen doch Auswirkungen auf praktische Fragestellungen, z.B. müssen Sie sich über solche prinzipielle Grenzen bewusst sein, wenn es um Programmanalysetool für die Software-Qualitätssicherung geht.

Wir werden die Berechenbarkeitstheorie in diesem Semester nur eher kurz gegen Ende des Semesters anschneiden.

## Komplexitätstheorie

Während es in der Berechenbarkeitstheorie darum geht, was prinzipiell möglich oder unmöglich ist, geht es bei der Komplexitätstheorie um die Dinge, die berechenbar sind. Die Fragestellung ist dann, welche Probleme davon effizient lösbar sind und für welche Probleme es keinen effizienten Algorithmus geben kann.

Dazu muss auch erst einmal geklärt werden, wie überhaupt die Komplexität/Effizienz von Algorithmen sinnvoll "gemessen", d.h. formal beschrieben werden kann.

Einige Grundlagen der Komplexitätstheorie werden Sie im nächsten Semester in der Vorlesung Algorithmen&Datenstrukturen kennenlernen.

## 1.2 Warum ist Theoretische Informatik wichtig?

---

An folgendem Beispiel aus dem Bereich der Programmiersprachen-Implementierung möchte ich darstellen, in welcher Weise Konzepte aus der theoretischen Informatik zur Lösung praktischer Probleme dienen.

### Grundlagen Compilerbau und Formale Sprachen

Haben Sie sich schon einmal Gedanken gemacht, wie ein Compiler Programme verarbeitet, z.B. das folgende Programmstück? Welche Teilaufgaben muss der Compiler dabei erledigen?



```

int zaehler;
int nenner;

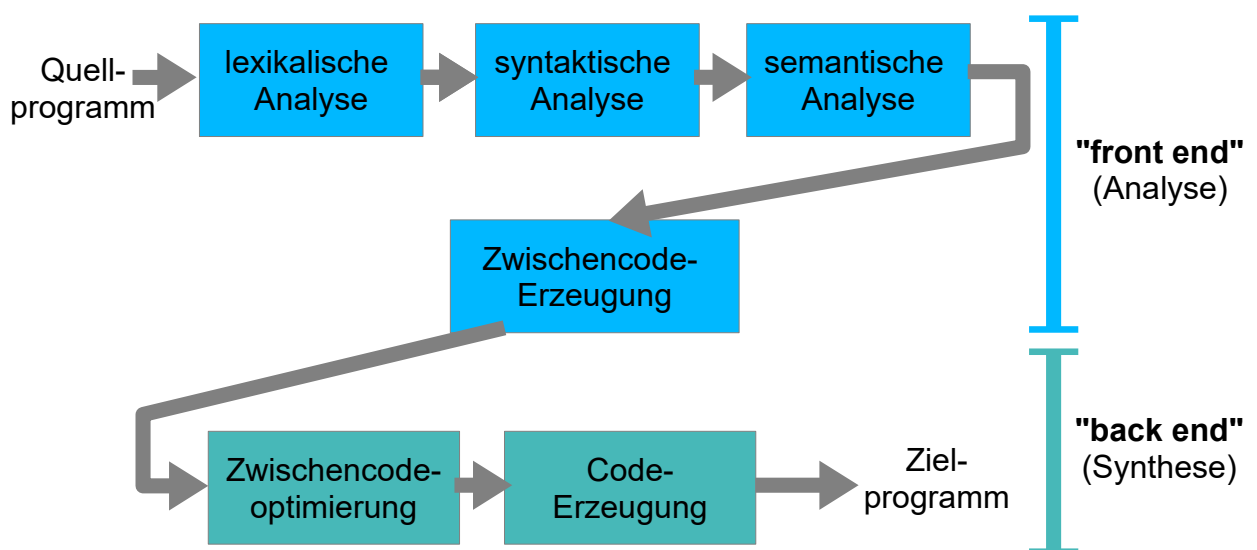
int ggtEuklid(int n, int m) {
    while (n != m) {
        if (n > m) n -= m;
        else      m -= n;
    }
    return n;
}

void kuerzen() {
    int gt = ggtEuklid(zaehler, nenner);
    zaehler /= gt;
    nenner /= gt;
}

```

## Prinzipieller Aufbau eines Compilers

Die Übersetzung eines Programms ist eine sehr schwierige und komplexe Problemstellung. Diese Aufgabe wird typischerweise in verschiedene Teilaufgaben ("Phasen") aufgeteilt, wie in folgendem Schaubild dargestellt:



Die ersten Phasen, das sog. "front end", prüfen, ob ein Programm syntaktisch korrekt aufgebaut ist und bereiten die Weiterverarbeitung vor. Das sog. "back end" ist dann für die eigentliche Übersetzung, d.h. die Codeerzeugung zuständig.

Die Phasen des *front ends* basieren vor allem auf Konzepten aus dem Bereich *Automatentheorie und formale Sprachen*, die in diesem Semester einen thematischen Schwerpunkt bilden.

## Lexikalische Analyse (sog. Scanner oder Lexer)

Die erste Phase im Compiler ist die sog. *lexikalische Analyse* (vom Präprozessor, wie z.B. bei C/C++, abgesehen). Aufgabe der lexikalischen Analyse ist die Zerlegung des Programmquelltexts in sog. *Token* (z.B. Zahlen, Bezeichner, spezielle Symbole, ...), d.h. elementare Bedeutung tragende Einheiten. Bedeutungslos Teile, wie Kommentare oder Leerzeichen, Zeilenumbrüche etc. werden dabei überlesen.


Ein Programm ist zunächst als Textdatei einfach eine Folge von Zeichen. Aus dieser Folge von Zeichen wird eine Folge von Token (unterschiedlicher Art) gebildet.

Für das Beispiel oben würde sich folgendes ergeben:

int   zaehler   ;   int   nenner   ;   int   ggtEuklid   (   int   ...

 Schlüsselwort-Token

 Bezeichner-Token

 sonst. Symbol-Token

Wird eine neue Programmiersprache definiert, muss festgelegt werden, wie solche Token aufgebaut sein dürfen. Die Umgangssprache ist zu unpräzise und eignet sich daher dazu schlecht. Hier hilft die theoretische Informatik, Teilgebiet Formale Sprachen:

- ▶ Durch sog. **reguläre Ausdrücke** kann eindeutig definiert werden, wie die Token, z.B. Bezeichner oder Zahlen, aufgebaut sein dürfen.
- ▶ In der lexikalischen Analyse müssen dann die einzelnen Token erkannt werden – und das möglichst effizient. Das geht mit einem weiteren Konzept aus dem Gebiet der formalen Sprachen, mit sog. **endlichen Automaten**.

Wenn Sie sich dann mit regulären Ausdrücken und endlichen Automaten auskennen, werden Sie feststellen, dass es sehr schwierig ist, einen passenden endlichen Automaten zu bilden, der die Token richtig erkennt. Aber auch hier hilft die Theorie der formalen Sprachen weiter: Reguläre Ausdrücke können mit einem geeigneten Transformationsverfahren systematisch in passende endliche Automaten "übersetzt" werden (und endliche Automaten sind sehr einfach und effizient zu implementieren).

Die Unterstützung durch die Theorie der Formalen Sprachen und Automaten geht aber noch weiter:

- ▶ **Scanner-Generator:** Da die Umwandlung zwischen regulären Ausdrücken und endlichen Automaten formal klar definiert ist, kann auch ein Programm diese im Prinzip nicht wirklich schwierige, aber sehr mühsame Aufgabe übernehmen. Solche Programme nennt man *Scanner-* oder *Lexer-Generatoren*.

## Beispiel 1.1 - Token-Spezifikation für Scanner-Generator

```
Number: (Digit)+ ( '.' (Digit)* (Exponent)? | Exponent)
...

Exponent:  ('e'|'E') ('+'|'-')? (Digit)+ ;

Digit:  '0'..'9';
```

Das Beispiel zeigt die Spezifikation für C++-Gleitpunktzahlen mittels regulärer Ausdrücke in der Notation für den Scannergenerator ANTLR (das müssen Sie jetzt noch nicht verstehen, es wird Ihnen am Ende des Semesters aber weitgehend verständlich sein, selbst wenn Sie die Details von ANTLR nicht kennen).

Durch einen einfachen "Knopfdruck", d.h. Aufruf des Scannergenerators mit dieser abstrakten Beschreibung als Input, kann der gesamte Code für die lexikalische Analyse generiert werden.

## Syntaktische Analyse (sog. Parser)

Die Aufgabe der zweiten Phase des Compilers, der syntaktischen Analyse, ist die Prüfung auf syntaktische Korrektheit entsprechend der Grammatik der Programmiersprache.

- ▶ Auch das setzt voraus, dass die Grammatik einer Programmiersprache exakt definiert ist. Die formale Methode dafür sind sog. **kontextfreie Grammatiken**.
- ▶ Sobald die Syntax einer Sprache mittels einer geeigneten kontextfreien Grammatik definiert ist, können dann passende **Syntaxanalyse-Verfahren** eingesetzt werden.

Auch hier hilft das Wissen aus der Theoretischen Informatik in der Praxis noch weiter: Der Programmcode für die syntaktische Analyse des Compilers muss nicht von Hand geschrieben werden. Da alles formal klar definiert ist, kann der Code auch von einem Programm "geschrieben", d.h. generiert werden:

- ▶ **Parser-Generator**: Programm, das den Programmcode für die syntaktische Analyse aus einer formalen Beschreibung der Grammatik generiert

## Beispiel 1.2 - Java-Grammatik

Syntax von Java-Anweisungen (vereinfacht) als Input für einen Parser-Generator:

```
statement ::=
    if ( condition ) statement
  | if ( condition ) statement else statement
  | while ( condition ) statement
  | do statement while ( expression ) ;
...
```

**Ergebnis der syntaktischen Analyse:** Die syntaktische Analyse liefert zum einen natürlich das Ergebnis, ob das Programm syntaktisch korrekt ist bzw. wo es Syntaxfehler enthält. Für die Weiterverarbeitung wird aber noch mehr gebraucht:

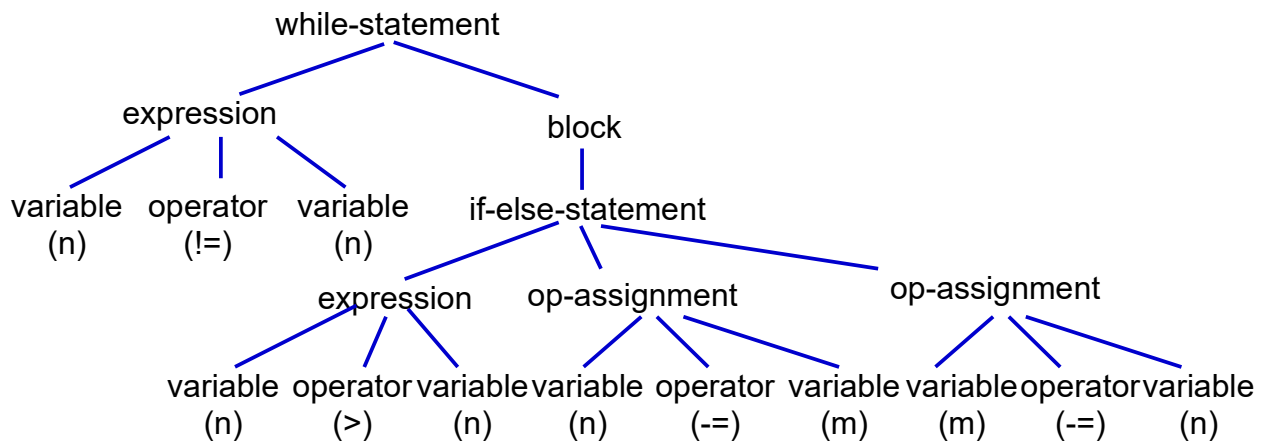
Typischerweise wird während der Syntaxanalyse eine interne Datenstruktur aufgebaut, die den strukturellen Aufbau des Programms beschreibt mit allen Informationen, die für die Codegenerierung bzw. Ausführung relevant sind, einen sog. *Abstrakten Syntaxbaum* (engl. *abstract syntax tree*, AST).

(*Bäume* sind ein Konzept aus der Graphentheorie, das in der Informatik sehr häufig verwendet wird).

Für die While-Schleife

```
while (n != m) {  
    if (n > m) n -= m;  
    else      m -= n;  
}
```

aus dem Programm oben könnte ein Abstrakter Syntaxbaum ungefähr so aussehen:



Auf die weiteren Phasen des Compiler soll hier nicht genauer eingegangen werden.

- ▶ Die *Semantische Analyse* macht weitere Prüfungen auf Basis des Abstrakten Syntaxbaums, z.B. wird hier die Typprüfung durchgeführt und es wird geprüft, ob bzw. wo verwendete Variablen, Methoden oder Klassen auch definiert wurden. Der Abstrakte Syntaxbaum wird dann um entsprechende Informationen (z.B. Typen) erweitert.

Das Backend besteht oft aus mehreren Phasen und unterschiedlichen Zwischendarstellungen des Programms, um verschiedenen Optimierungen durchführen zu können, bevor am Ende der generierte Code geschrieben wird.

## Fazit

Der Bereich Compilerbau zeigt beispielhaft sehr schön, welche Rolle die

Theoretische Informatik spielt:

- ▶ Um ein schwieriges Probleme zu lösen, muss es erst einmal auf abstrakter Ebene formalisiert werden (z.B. Wie definiere ich eine Sprache? Wie analysiere ich grammatikalisch eine Sprache?).
- ▶ Wird auf abstrakter Ebene eine Lösung solcher Probleme gefunden (z.B. Wie kann eine effiziente Syntaxanalyse durchgeführt werden?), kann die abstrakte Problemlösung dann relativ leicht wieder auf Programmiersprachen-Ebene umgesetzt und implementiert werden.
- ▶ Teilweise kann durch das Wissen der formalen Methoden die recht mühsame Umsetzung von der abstrakten Ebene auf die Programmiersprachenebene durch Programme (Generatoren) komplett übernommen werden.

## 1.2.1 Denkweisen der Informatik

---

### *Informatiker ↔ Programmierer*

Fast jeder Studiengang in der Hochschule beinhaltet auch ein, zwei Programmierkurse. D.h. programmieren kann eigentlich jeder, der ein technisch orientiertes Studium hat. Was zeichnet Sie als Informatiker später dann überhaupt im Vergleich zu anderen Leuten aus, die auch Programmieren können?

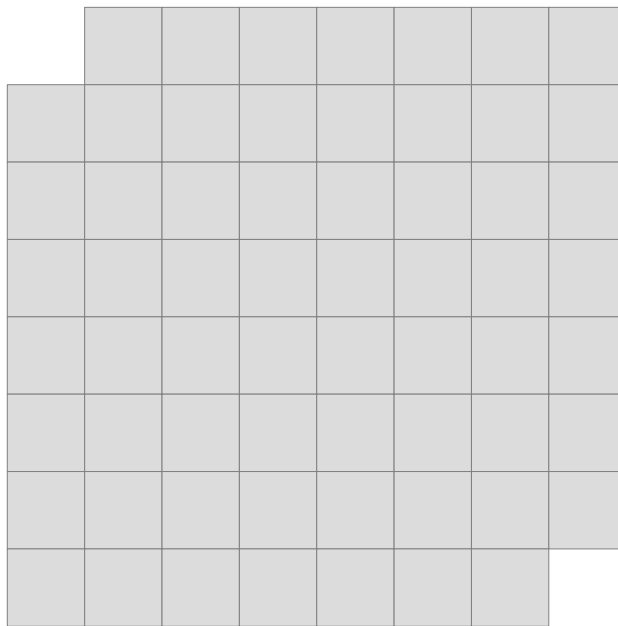
Jedes Fachgebiet hat seine spezifischen **Denkweisen**. Ein wesentlicher Unterschied, der Informatiker auszeichnet, ist die Denkweise, d.h. wie sie an Probleme herangehen und welche Lösungsansätze und -methoden sie im Blick haben. Theoretische Informatik (und natürlich auch andere Vorlesungen) soll solche Denkweisen schulen und dafür passende formale Konzepte vermitteln – das ist also auch gewissermaßen "Brain-Jogging" für Sie.

Zum Abschluss dieses Abschnitts noch ein kleine Knobelaufgabe als "Brain-Jogging" (auch wenn zunächst nicht direkt Informatik-bezogen).

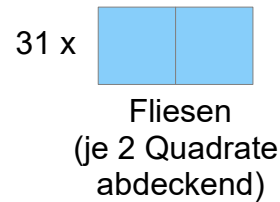
### *Aufgabe 1.3 - "Problem des Fliesenlegers"*

Ein Fliesenleger soll in einem Raum den Boden mit Fliesen belegen:

- ▶ Die Bodenfläche besteht aus quadratischen Feldern, die in 8 Reihen und 8 Spalten angeordnet sind. An zwei gegenüberliegende Ecken fehlt ein Quadrat, d.h. insgesamt sind 62 Quadrate zu belegen.



Raum  
(62 Quadrat)



- ▶ Es stehen 31 Fliesensteine zur Verfügung, die jeweils genau zwei Quadrate groß sind. Die Fliesen können in vertikaler oder horizontaler Richtung gelegt werden.

**Problemstellung:** Der Fliesenleger hat leider seinen Fliesenschneider vergessen, so dass er die Fliesen nicht halbieren kann. Kann er alle 62 Felder mit den 31 ganzen Fliesen abdecken? Wie muss er ggf. die Fliesen legen?

Wie würden Sie als Informatiker an diese Fragestellung herangehen?

## 1.3 Ziele und Inhalt der Lehrveranstaltung

### Lernziele

- ▶ Grundlegende Begriffe der Theoretischen Informatik kennen (z.B. Graphen, Sprachen, Grammatiken, Automaten, ...)
- ▶ Grundlegende Beschreibungs- und Analyseverfahren der Informatik verstehen und anwenden können (z.B. reguläre Ausdrücke und kontextfreie Grammatiken, prädikatenlogische Formeln. Textanalyse mit endlichen Automaten, Top-Down-Syntaxanalyse).
- ▶ Typische Denkmuster für die Problemanalyse erlernen (z.B. Konzept der Invarianten)
- ▶ Grundlegende Problemklassen in Anwendungsproblemen erkennen (z.B. was kann mit regulären Ausdrücken beschrieben werden, wann werden aber kontextfreie Grammatiken benötigt)

- ▶ Mit formalen Systemen sinnvoll umgehen können (z.B. reguläre Ausdrücke, Grammatiken oder prädikatenlogische Formeln äquivalent umformen und vereinfachen können)

## Inhalt

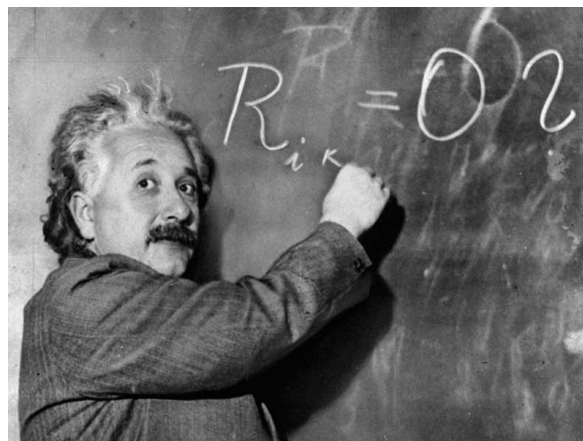
Folgende Themen sind für dieses Semester geplant:

- (1) Mathematische Grundlagen
- (2) Grundbegriffe der Graphentheorie
- (3) Formale Sprachen
- (4) Reguläre Ausdrücke
- (5) Kontextfreie Grammatiken
- (6) Deterministische endliche Automaten
- (7) Nichtdeterministische endliche Automaten
- (8) Kellerautomaten
- (9) Effiziente Top-Down-Syntaxanalyse
- (10) Einführung in die Prädikatenlogik
- (11) Berechenbarkeit und Unentscheidbarkeit

## Motto

Es gibt nichts Praktischeres als eine gute Theorie!

[Kurt Lewin]



## 1.4 Infos und Literatur

---

### Moodle-Kurs

- ▶ <http://moodle.hs-ulm.de> (Fakultäten->Informatik->Informatik)

- ▶ Einschreibeschlüssel: **tiny2**
- ▶ Lektionen, Quizaufgaben, Aufgabenblätter, alte Klausuren, sonstige Infos

## Bücher

- ▶ D. W. Hoffmann: *Theoretische Informatik*, Hanser-Verlag, 2009, 39,90€  
Auch als E-Book in der Bibliothek
- ▶ R. Socher: *Theoretische Grundlagen der Informatik*. 3. Auflage, 2007 Hanser Verlag, 24,90€
- ▶ J.E. Hopcroft, R.M. Motwani, J.E. Ullman: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. 2. Auflage, 2002, Pearson Studium, 39,95€.  
Ein internationales Standardwerk. Tiefgehend und umfassend.
- ▶ M. Sipser: *Introduction to the Theory of Computation*. 2. Auflage, Thomson, 2006  
Ein internationales Standardwerk. Eher mathematisch-formal, nicht ganz so leicht zugänglich.
- ▶ Vossen/Witt: *Grundkurs Theoretische Informatik*. 4. Auflage, Vieweg-Verlag, 2006
- ▶ U. Schöning: *Theoretische Informatik – kurzgefasst*. 4. Auflage, 2001.  
Spektrum Akademischer Verlag, 20,00€
- ▶ G. Teschl, S. Teschl: *Mathematik für Informatiker*, Band 1. 3. Auflage, 2008, Springer (Kap. 5, 15, 16)
- ▶ P. Tittmann: *Graphentheorie*. Fachbuchverlag Leipzig, 2003
- ▶ A.V. Aho, M.S. Lam, R. Sethi, J.D. Ullman: *Compiler*, 2. akt. Auflage. Pearson Studium, 2008. Das "Drachenbuch" - die Compilerbau-Bibel!
- ▶ U. Schöning: *Logik für Informatiker*. 5. Auflage, Spektrum Akademischer Verlag, 2000

## Sonstiges

Da Theoretische Informatik ein etabliertes Standard-Lehrgebiet der Informatik ist, finden Sie im Internet an vielen Stellen (Youtube, Wikipedia, Google, ...).  
Informationen zu den einzelnen Themen

Beachten Sie, dass alle zur Verfügung gestellten Unterlagen dem **Urheberrecht** unterliegen. Sie dürfen diese Unterlagen nur für den eigenen Gebrauch im Rahmen der Vorlesung verwenden. Eine Weitergabe oder Veröffentlichung (Internet, soziale Medien, ...) ist nicht erlaubt.



*Diese Fragen sollten Sie nun beantworten können*

- ▶ Wie ist die Veranstaltung in diesem Semester organisiert?
- ▶ Wie viele Quizzes und Aufgabenblätter müssen für Sie für die Studienleistung (Schein) erfolgreich bearbeiten?
- ▶ Welche Teilgebiete gehören zur Theoretischen Informatik?
- ▶ Welche Funktion hat das Gebiet "Theoretische Informatik" innerhalb der Informatik?
- ▶ Wie werden Konzepte der Theoretischen Informatik bei der Definition und Implementierung von Programmiersprachen angewendet?