

Institut für Informatik, AG Theoretische Informatik  
Universität Göttingen

# Theoretische Informatik

## Vorlesung

Prof. Dr. Florin Manea

15. April 2025



## Allgemeine Informationen

# Wer?

Prof. Dr. Florin Manea:



- Doktorat @ Uni Bukarest (2007):  
Berechenbarkeits- und Komplexitätstheorie für biologisch inspirierte  
Berechnungsmodelle.



# Wer?

Prof. Dr. Florin Manea:



- ▶ Lektor – Bukarest Uni: Grundlagen der Informatik (2007–2009).

# Wer?

Prof. Dr. Florin Manea:



- ▶ Lektor – Bukarest Uni: Grundlagen der Informatik (2007–2009).
- ▶ Postdoc, Humboldt Stipendiat – Magdeburg Uni (2009–2011): Berechenbarkeits- und Komplexitätstheorie für biologisch inspirierte Berechnungsmodelle.



- ▶ Lektor – Bukarest Uni: Grundlagen der Informatik (2007–2009).
- ▶ Postdoc, Humboldt Stipendiat – Magdeburg Uni (2009–2011): Berechenbarkeits- und Komplexitätstheorie für biologisch inspirierte Berechnungsmodelle.
- ▶ Postdoc, Zuverlässige Systeme AG, Kiel Uni: Kombinatorik und Algorithmen auf Wörtern (2011–2013).



- ▶ Lektor – Bukarest Uni: Grundlagen der Informatik (2007–2009).
- ▶ Postdoc, Humboldt Stipendiat – Magdeburg Uni (2009–2011): Berechenbarkeits- und Komplexitätstheorie für biologisch inspirierte Berechnungsmodelle.
- ▶ Postdoc, Zuverlässige Systeme AG, Kiel Uni: Kombinatorik und Algorithmen auf Wörtern (2011–2013).
- ▶ DFG Eigene Stelle, Kiel Uni: Algorithmen (2013–2019).



- ▶ Lektor – Bukarest Uni: Grundlagen der Informatik (2007–2009).
- ▶ Postdoc, Humboldt Stipendiat – Magdeburg Uni (2009–2011): Berechenbarkeits- und Komplexitätstheorie für biologisch inspirierte Berechnungsmodelle.
- ▶ Postdoc, Zuverlässige Systeme AG, Kiel Uni: Kombinatorik und Algorithmen auf Wörtern (2011–2013).
- ▶ DFG Eigene Stelle, Kiel Uni: Algorithmen (2013–2019).
- ▶ Professor, Göttingen Uni: W2-Grundlagen der Informatik (2019 – 2022)  
Professor, Göttingen Uni: W3-Theoretische Informatik (2022 – )  
Website: <https://www.uni-goettingen.de/de/618738.html>

Übungen: **Dr. Tore Koß** und Tutorenteam.

# Was, Wann und Wo?



Theoretische Informatik – Vorlesung in Präsenz

## Theoretische Informatik – Vorlesung in Präsenz

- ▶ Vorlesungen: Präsenz, einmal pro Woche, jeden Dienstag zwischen 12:15 und 13:45.
- ▶ Die Vorlesung wird gemeinsam von mir und Dr. Tore Koß gehalten.

Alle Ankündigungen zur Vorlesung erfolgen über Mails in StudIP und Nachrichten im Chat-Raum der Vorlesung!

Bitte schauen Sie auch regelmäßig auf der StudIP-Seite der Vorlesung vorbei!

## Theoretische Informatik – Vorlesung in Präsenz

- ▶ Vorlesungsbezogene Uploads in Stud.IP:  
die Folien der Vorlesung (2025) +  
die Folien der Vorlesung (2024) +  
Handouts der Folien des Vortrags (2024) +  
das Skript der Vorlesung (2024).
- ▶ Ankündigungen, Fragen, Antworten:  
<https://chat.uni-goettingen.de/#/room/#TCS2025:uni-goettingen.de>

Das gesamte Material für diese Vorlesung wurde in den vergangenen Jahren von Prof. Carsten Damm entwickelt. Wir werden einige kleinere Änderungen an den Folien vornehmen, aber das Material ist im Wesentlichen das vom letzten Jahr.

# Was, Wann und Wo?



Theoretische Informatik – Übungen (ab nächster Woche!), **Dr. Tore Koß** und Tutorenteam.

- ▶ In der Vorlesungszeit wird jeweils am Wochenanfang ein neues Übungsblatt erscheinen mit i.d.R. vier Aufgaben:
  - ▶ die meisten betreffen bereits behandelte Themen
  - ▶ manche greifen absichtlich etwas vor, um die nächsten Themen vorzubereiten
  - ▶ Lösungen sind bis Anfang der Folgewoche im PDF-Format über das Vips-Modul von Stud.IP einzureichen.
- ▶ Idealerweise erzeugen Sie Ihre Abgaben mit LaTeX, Markdown o.ä. Bei handschriftlichen Lösungen (am besten eingescannt) achten Sie bitte besonders auf Lesbarkeit und klare Strukturierung.
- ▶ Die korrigierten Lösungen und erreichten Punkte werden über Vips erfasst. Als Prüfungsvorleistung müssen Sie mindestens 50% der erreichbaren Punkte erhalten sowie eine Ihrer Lösungen in einer der Übungen präsentieren.

In den Tutorien präsentieren Teilnehmer unter Anleitung erfahrener Studierender eigene Lösungen. Einmal Vorrechnen ist Pflicht, aber es ist sinnvoll, das mehrmals zu tun.



## Theoretische Informatik – Prüfung

### ► Schriftliche (Open-Book-)Klausur:

1. Termin: 15.8.2025, E-Prüfungsraum (Blauer Turm), 9:00 - 11:00
2. Termin: 18.9.2025, E-Prüfungsraum (Blauer Turm), 9:00 - 11:00.

Open-Book = das Skript und die Folien der Vorlesungen werden auf den Computern im Raum verfügbar sein.

- Inhalt: Themen im Zusammenhang mit den theoretischen Aspekten der Vorlesungen und den Übungen.
- Als Prüfungsvorleistung müssen Sie mindestens 50% der erreichbaren Punkte erhalten sowie eine Ihrer Lösungen in einer der Übungen präsentieren.

Sehr wichtig:

- ▶ Für die Übungsteilnahme müssen Sie sich bei Flexnow für **B.Inf.1201.Ue** anmelden (Anmeldefrist bis zum Ende der Vorlesungszeit – 18.07.2025).

## Anmeldung im FlexNow:

Um die Prüfungszulassung zu erhalten (und für zukünftige Semester zu behalten, falls Sie die Prüfung nicht in diesem Semester schreiben wollen), melden Sie sich bitte zu den TI Übungen in FlexNow an: **B.Inf.1201.Ue**

- ▶ Hauptthemen:
  - ▶ verschiedene Berechnungsmodelle (Computing Models), von strukturell einfachen (endliche Automaten) bis hin zu komplexen (Turingmaschinen, äquivalent zu modernen Computers);
  - ▶ der Begriff der Berechnung (für verschiedene Berechnungsmodelle) und eine Analyse dessen, was wir mit den jeweiligen Modellen berechnen können.
  - ▶ Chomskys Hierarchie der formalen Sprachen, Grammatiken, die formale Sprachen generieren, und Automaten, die diese akzeptieren.
  - ▶ Anwendungen und Verbindungen des Obigen zu praktischen Themen (Parsing, Natural Language Processing, etc.).

## Das heißt:

In diesem Kurs werden wir versuchen, Folgendes zu erreichen:

- ▶ Ein besseres Verständnis dafür, was Computing bedeutet.
- ▶ Ein besseres Verständnis der Grundlagen der Informatik und der modernen Computer.
- ▶ Ein besseres Verständnis dafür, welche Herausforderungen bei der Konstruktion von Compilern oder Anwendungen der natürlichen Sprachverarbeitung auftreten können.

Am Ende der Vorlesung werden die Studenten idealerweise in der Lage sein, die Idee der Berechenbarkeit (was kann auf einem bestimmten Berechnungsmodell berechnet werden), die Grenzen der Berechenbarkeit (gibt es Funktionen, die nicht berechnet werden können?) und den Zusammenhang zwischen formalen Sprachen und Berechenbarkeit besser zu verstehen.

# Theoretische Informatik

Florin Manea  
(basierend auf den Folien von Carsten Damm)

Stand: 14. April 2025

# Inhaltsüberblick

- 1 Einführung
- 2 Endliche Automaten und reguläre Sprachen
- 3 Kontextfreie Sprachen und Kellerautomaten
- 4 Typ1 und Typ0-Sprachen, Turing-Maschinen
- 5 Berechenbarkeit

1 Einführung

2 Endliche Automaten und reguläre Sprachen

3 Kontextfreie Sprachen und Kellerautomaten

4 Typ1 und Typ0-Sprachen, Turing-Maschinen

5 Berechenbarkeit

## 1 Einführung

- Bücher
- Motivation
- Symbole, Wörter und Sprachen
- Operationen auf Sprachen
- Sprachklassen
- Reguläre Ausdrücke

# Jetzt:

## 1 Einführung

- Bücher
- Motivation
- Symbole, Wörter und Sprachen
- Operationen auf Sprachen
- Sprachklassen
- Reguläre Ausdrücke

# Bücher

Uwe Schöning: Theoretische Informatik — kurz gefasst



deckt den kompletten Vorlesungsstoff ab, kommt schnell zur Sache,  
gut lesbar

# Bücher

Uwe Schöning: Theoretische Informatik — kurz gefasst



deckt den kompletten Vorlesungsstoff ab, kommt schnell zur Sache,  
gut lesbar

Dirk W. Hoffmann: Theoretische Informatik



etwas andere Themenauswahl (das Wichtigste ist aber dabei), an-  
schaulich und weniger formal

# Bücher

Uwe Schöning: Theoretische Informatik — kurz gefasst



deckt den kompletten Vorlesungsstoff ab, kommt schnell zur Sache,  
gut lesbar

Dirk W. Hoffmann: Theoretische Informatik



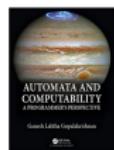
etwas andere Themenauswahl (das Wichtigste ist aber dabei), an-  
schaulich und weniger formal

Das beste Buch für diese Vorlesung:

Hopcroft, John E.; Ullman, Jeffrey D. (1979). Introduction to Automata Theory, Languages, and Computation (1st ed.). Addison-Wesley. ISBN 0-201-02988-X.

# Python und Theorie?

Bemerkung zu „Automata and Computability — A programmer’s perspective“ von Gopalakrishnan



- ist ziemlich informal, daher *als alleinige Grundlage nicht geeignet*
- aber innovativ: Erläuterung des gesamten Stoffs anhand von **Python-Experimenten** (mit der dafür entwickelten Bibliothek *Jove*)

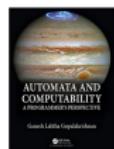
Einige Beispiele im Skript werden mit Python-Code dargestellt/unterstützt!

Muss ich Python können, um dem Kurs zu folgen?

Nein! Python wird hier benutzt, Algorithmen verständlich aber präzise zu beschreiben. Wer *irgendeine* imperativen Programmiersprache kennt, kann auch diesen Python-Code verstehen.

# Python und Theorie?

Bemerkung zu „Automata and Computability — A programmer's perspective“ von Gopalakrishnan



- ist ziemlich informal, daher *als alleinige Grundlage nicht geeignet*
- aber innovativ: Erläuterung des gesamten Stoffs anhand von **Python-Experimenten** (mit der dafür entwickelten Bibliothek *Jove*)

Einige Beispiele im Skript werden mit Python-Code dargestellt/unterstützt!

Muss ich Python können, um dem Kurs zu folgen?

Nein! Python wird hier benutzt, Algorithmen verständlich aber präzise zu beschreiben. Wer *irgendeine* imperativen Programmiersprache kennt, kann auch diesen Python-Code verstehen.

Kann/muss ich in diesem Kurs Python lernen?

Nein. Es wird kaum etwas zu Python vermittelt — das ist auch kein Prüfungsstoff.

# Jetzt:

## 1 Einführung

- Bücher
- Motivation
- Symbole, Wörter und Sprachen
- Operationen auf Sprachen
- Sprachklassen
- Reguläre Ausdrücke

# Syntaxbeschreibungen

- *Bezeichner* bestehen aus Buchstabe, optional gefolgt von Ziffern, und Buchstaben. Beispiel: Head1\_ptr

# Syntaxbeschreibungen

- *Bezeichner* bestehen aus Buchstabe, optional gefolgt von Ziffern, und Buchstaben. Beispiel: Head1\_ptr
- *Variablen-deklarationen* bestehen aus Typ und einem (oder Komma-getrennt mehreren) Variablenbezeichner(n) sowie ;. Beispiel: int a, b, c;

# Syntaxbeschreibungen

- *Bezeichner* bestehen aus Buchstabe, optional gefolgt von Ziffern, und Buchstaben. Beispiel: Head1\_ptr
- *Variablen-deklarationen* bestehen aus Typ und einem (oder Komma-getrennt mehreren) Variablenbezeichner(n) sowie ;. Beispiel: int a, b, c;
- *dezimales Gleitkommaliteral* besteht aus ganzem Teil, Dezimalpunkt, gebrochenem Teil (wobei höchstens eins der beiden Teile fehlen darf) optional gefolgt von E oder e mit Exponententeil. Die Teile bestehen aus Ziffern, ganzem und Exponententeil darf ein Vorzeichen vorangestellt sein. Beispiele: -1.E-45, +0.1

# Syntaxbeschreibungen

- *Bezeichner* bestehen aus Buchstabe, optional gefolgt von Ziffern, und Buchstaben. Beispiel: Head1\_ptr
- *Variablen-deklarationen* bestehen aus Typ und einem (oder Komma-getrennt mehreren) Variablenbezeichner(n) sowie ;. Beispiel: int a, b, c;
- *dezimales Gleitkommaliteral* besteht aus ganzem Teil, Dezimalpunkt, gebrochenem Teil (wobei höchstens eins der beiden Teile fehlen darf) optional gefolgt von E oder e mit Exponententeil. Die Teile bestehen aus Ziffern, ganzem und Exponententeil darf ein Vorzeichen vorangestellt sein. Beispiele: -1.E-45, +0.1
- *geschachtelter Programmblock* besteht aus korrekter Klammerung (wie {{}}{{}}}) sowie Bestandteilen zwischen den Klammern.

# Syntaxbeschreibungen

- *Bezeichner* bestehen aus Buchstabe, optional gefolgt von Ziffern, und Buchstaben. Beispiel: Head1\_ptr
- *Variablen-deklarationen* bestehen aus Typ und einem (oder Komma-getrennt mehreren) Variablenbezeichner(n) sowie ;. Beispiel: int a, b, c;
- *dezimales Gleitkommaliteral* besteht aus ganzem Teil, Dezimalpunkt, gebrochenem Teil (wobei höchstens eins der beiden Teile fehlen darf) optional gefolgt von E oder e mit Exponententeil. Die Teile bestehen aus Ziffern, ganzem und Exponententeil darf ein Vorzeichen vorangestellt sein. Beispiele: -1.E-45, +0.1
- *geschachtelter Programmblock* besteht aus korrekter Klammerung (wie {{}}{{}}}) sowie Bestandteilen zwischen den Klammern.
- Prototyp und Funktionsdefinition müssen zusammenpassen, wie hier:

```
char sq(int);
char sq(int a) { return a*a; }
```

# Compilation in den Pioniertagen der Programmierung

Das war laut Donald Knuth eine obskure und fehleranfällige Trickkiste.

Z.B. um Operatorpräzedenz \* vor + zu erreichen:

- ① ersetze jedes + durch )))+((( und jedes \* durch )))\*((
- ② balanciere den Ausdruck mit genügend zusätzlichen Klammern

# Compilation in den Pioniertagen der Programmierung

Das war laut Donald Knuth eine obskure und fehleranfällige Trickkiste.

Z.B. um Operatorpräzedenz \* vor + zu erreichen:

- ① ersetze jedes + durch )))+((( und jedes \* durch )))\*((
- ② balanciere den Ausdruck mit genügend zusätzlichen Klammern

## Beispiel

- 1 + 2 \* 3
- 1 )))+((( 2 ))\*(( 3
- ((( 1 )))+((( 2 ))\*(( 3 )))

# Vier programmierrelevante Sprachstrukturen

## Reguläre Muster

- endliche aber prinzipiell beliebige Länge (Beispiel: Bezeichner)
- Wiederholungen (Beispiel: Variablendeklarationen)
- Kombinationen (Beispiel: endlich viele Varianten bei GK-Literalen)

Typisch regulär: 0-Folgen (beliebiger Länge) gefolgt von 1

## Kontextfreie Muster

- Klammerungen (Beispiel: Klammern bei Programmblöcken und Ausdrücken)
- korrekte Klammerung kann „für sich“ geprüft werden, hängt nicht vom Kontext ab

Typisch kontextfrei: *Palindrome* = Zeichenfolgen, die vorwärts wie rückwärts gelesen gleich sind

# Vier programmierrelevante Sprachstrukturen (Forts.)

## Kontextsensitive Muster

- Bedingungen (Beispiele: Prototyp vor Funktion, Variable vor Benutzung deklarieren)

Typisch kontextsensitiv: 0/1-Folgen der Form 01001...01  $\underbrace{00\dots0}_{n-1}$  1  $\underbrace{00\dots0}_{n} 1$ .

## Allgemeine Muster

- an beliebige Berechnungen geknüpfte Bedingungen (Beispiel:  
Input/Output-Paare wie Zahlenfolge/sortierte Zahlenfolge)

Beispiel: Paare  $(M, w)$  aus Programmtext  $M$  und Eingabe  $w$ , so dass  $M$  auf der Eingabe  $w$  nicht in eine Endlosschleife gerät

# Vier programmierrelevante Sprachstrukturen (Forts.)

## Muster definieren Sprachen

Die Menge der Wörter, die einem bestimmten Muster entsprechen, ist eine *formale Sprache*. Wir definieren später ganz exakt, was formale und speziell reguläre, kontextfreie, kontextsensitive und rekursiv aufzählbare Sprachen sind.

# Parsen

## Was macht ein Parser?

(engl. to parse: „analysieren“, bzw. lateinisch pars: „Teil“) = Algorithmus für die Zerlegung/Umwandlung einer Eingabe in ein weiterverarbeitbares Format

# Parsen

## Was macht ein Parser?

(engl. to parse: „analysieren“, bzw. lateinisch pars: „Teil“) = Algorithmus für die Zerlegung/Umwandlung einer Eingabe in ein weiterverarbeitbares Format

## Parser für

- reguläre Muster: *endliche Automaten* (**FA** - finite automaton)  
Rabin & Scott (Ende 1950er)
- kontextfreie Muster: *Kellerautomat* (**PDA** - pushdown automaton)  
Ginsburg & Greibach (Anfang 1960er)
- kontextsensitive Muster: *linear beschränkte Automaten* (**LBA**)  
Kuroda (Mitte 1960er)

FA, PDA, LBA sind vereinfachte Varianten von Turing-Maschinen (TM).

# Parsen

## Was macht ein Parser?

(engl. to parse: „analysieren“, bzw. lateinisch pars: „Teil“) = Algorithmus für die Zerlegung/Umwandlung einer Eingabe in ein weiterverarbeitbares Format

## Parser für

- reguläre Muster: *endliche Automaten* (**FA** - finite automaton)  
Rabin & Scott (Ende 1950er)
- kontextfreie Muster: *Kellerautomat* (**PDA** - pushdown automaton)  
Ginsburg & Greibach (Anfang 1960er)
- kontextsensitive Muster: *linear beschränkte Automaten* (**LBA**)  
Kuroda (Mitte 1960er)

FA, PDA, LBA sind vereinfachte Varianten von Turing-Maschinen (TM).

## Parser für allgemeine Muster?

Es existiert kein allgemeines Verfahren, solche Parser zu konstruieren — das ist sogar unmöglich!

# Parsen

## Was macht ein Parser?

(engl. to parse: „analysieren“, bzw. lateinisch pars: „Teil“) = Algorithmus für die Zerlegung/Umwandlung einer Eingabe in ein weiterverarbeitbares Format

## Parser für

- reguläre Muster: *endliche Automaten* (**FA** - finite automaton)  
Rabin & Scott (Ende 1950er)
- kontextfreie Muster: *Kellerautomat* (**PDA** - pushdown automaton)  
Ginsburg & Greibach (Anfang 1960er)
- kontextsensitive Muster: *linear beschränkte Automaten* (**LBA**)  
Kuroda (Mitte 1960er)

FA, PDA, LBA sind vereinfachte Varianten von Turing-Maschinen (TM).

## Parser für allgemeine Muster?

Es existiert kein allgemeines Verfahren, solche Parser zu konstruieren — das ist sogar unmöglich! Beispiel: Es ist algorithmisch *unmöglich* zu entscheiden, ob eine gegebene Eingabe ein beliebig gegebenes Programm in eine Endlosschleife bringt.