DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

THEOPLUS

Ergänzende Inhalte der theoretischen Informatik

Roman Wetenkamp

5. Mai 2022

Randnotizen

Dieses Skript ist das Ergebnis meiner Beschäftigung mit weiteren Inhalten der theoretischen Informatik, die über die Vorlesungen zu theoretischer Informatik an der DHBW hinausgehen. Ich möchte mit diesem Skript meine Ergebnisse sichern, für andere nutzbar und zugänglich machen und mich didaktisch ausprobieren. Bei der Auswahl der Inhalte habe ich mich im wesentlichen an den Inhalten der Quellen [1], [2] und [3] und dem Modulhandbuch der DHBW Mannheim für den Studiengang „Informatik – Cyber Security“ orientiert.

Mitwirken Für Fragen, Anmerkungen und Kritik bin ich jederzeit sehr offen. Ich freue mich über E-Mails an s200376@student.dhbw-mannheim.de oder über Issues / Pull Requests im zu diesem Skript dazugehörigen GitHub-Repository.

Inhaltsübersicht

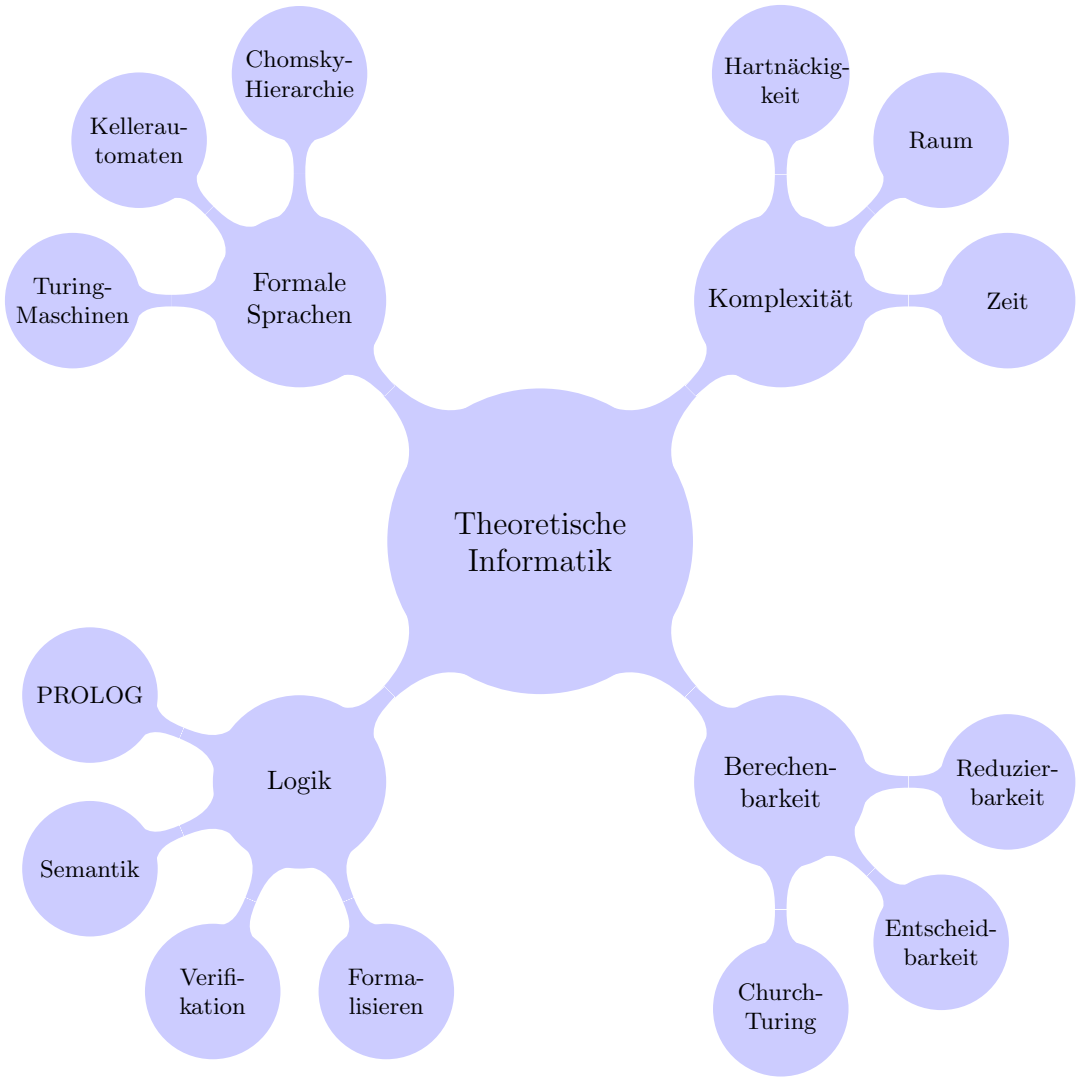

Abbildung 0.1.: Inhaltsübersicht. Eigene Darstellung.

Inhaltsverzeichnis

Teil I.

Formale Sprachen

1. Sprachklassen

Motivation Sprachen sind faszinierend: In unseren ersten Lebensjahren lernen wir die Sprache(n) unserer Eltern, noch so intuitiv wie nie wieder. In der Schule kommen Fremdsprachen hinzu, die wir mal mehr, mal weniger gerne lernen. Als Jugendliche experimentieren wir mit der Sprache, verändern sie, um uns von Erwachsenen abzugrenzen, nutzen Sie als Ausdruck unserer Lebenswirklichkeit. Später lernen Sie Programmiersprachen, fügen sie vielleicht scherzhaft der Liste Ihrer Sprachkenntnisse hinzu. Doch das ist weit mehr als ein plumper Nerdwitz: Diese und jene Sprachen eint viel!

Definition 1.0.1.
Eine (formale) Sprache ist eine Menge von Zeichenketten, die aus den Symbolen eines beliebigen Alphabets aufgebaut sind. [vgl. 1, S. 1]

Wenn Sie an das Sprachenlernen im schulischen Kontext denken, vergleichen Sie diese Definition vielleicht mit dem Vokabellernen. Demnach wäre Englisch die Menge aller englischen Wörter. Das erscheint uns falsch: Die bloße Kenntnis aller englischen Begriffe lässt Sie noch lange kein korrektes Englisch sprechen. Dafür gibt es Grammatiken – Regelwerke, die aussagen, wann ein Satz korrekt ist. Mit ihnen wollen wir uns nun aus formaler Sicht näher beschäftigen.

1.1. Die Chomsky-Hierarchie

In Ihrer Vorlesung haben Sie gelernt, dass es verschiedene Typen von Grammatiken gibt. Höchstwahrscheinlich wurden kontextfreie Grammatiken eingeführt – Diese Klassen erweitern wir nun entsprechend einer Hierarchie von Noam Chomsky, der . Dieser Aufbau verläuft analog zu den Zahlenräumen: Die natürlichen Zahlen $\mathbb{N}$, die Sie zum Zählen nutzen, ergeben zusammen mit ihren negativen Entsprechungen und der Null die ganzen Zahlen $\mathbb{Z}$, dann definieren Sie unendlich viele Zahlen dazwischen und nennen sie die rationalen Zahlen $\mathbb{Q}$. Wozu nach endlich vielen Nachkommastellen aufhören? Sie kommen zu $\mathbb{R}$ und schließlich zu $\mathbb{C}$. Analog zu dieser Teilmengenbeziehungen konstruieren wir nun auch die Sprachklassen der Chomsky-Hierarchie.

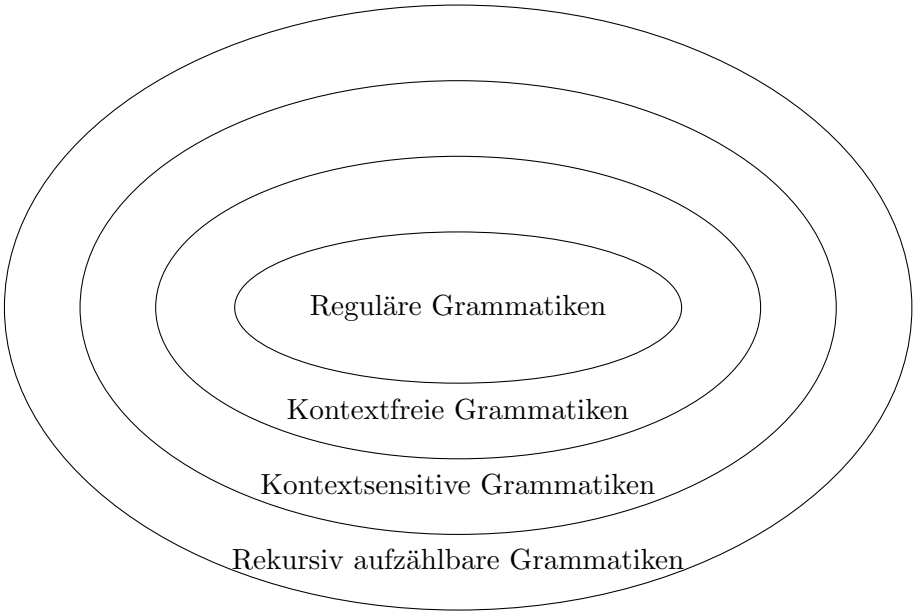

Abbildung 1.1.: Chomsky-Hierarchie formaler Sprachen. Eigene Darstellung.

Bemerkung 1. Eine reguläre Grammatik ist folglich auch eine kontextfreie, kontextsensitive und rekursiv aufzählbare Grammatik, unterliegt jedoch noch einigen weiteren Bedingungen und Beschränkungen als ihre Oberklassen.

Nacheinander – von innen nach außen – werden wir nun diese Sprachklassen definieren.

1.2. Typ 3: Reguläre Grammatiken

Zu Beginn dieses Themengebiets der Theoretischen Informatik widmeten wir uns Automaten. Wir unterschieden zunächst deterministische endliche (DEA, *engl.:* DFA) und nicht-deterministische endliche Automaten (NEA, *engl.:* NFA). Ebenso definierten wir reguläre Sprachen und stellten die Äquivalenz der drei Objekte fest. Wir werden reguläre Grammatiken anhand der nächstgrößeren Klasse, der kontextfreien Grammatiken, definieren. So werden reguläre Grammatiken zu kontextfreien Grammatiken, die spezielle Bedingungen erfüllen.

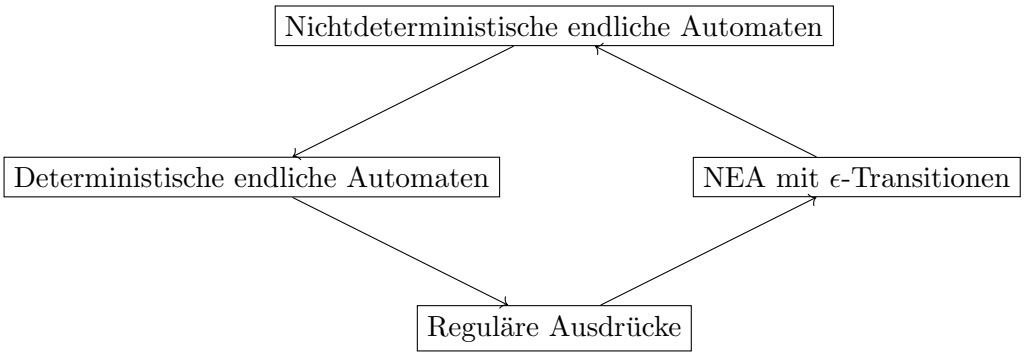

Abbildung 1.2.: Äquivalenzkonstruktionen (Quelle: [1, S. 30])

Definition 1.2.1.
Eine **kontextfreie Grammatik** G ist ein 4-Tupel $G = (V, T, P, S)$, wobei gilt:

- V bezeichnet eine Menge von Variablen.
- T bezeichnet eine Menge von Terminalen.
- P bezeichnet eine Menge von Produktionen (also Regeln), wobei eine Produktion A der Form $A \to \alpha$ mit $\alpha \in (V \cup T)*$ ist.
- S bezeichnet ein Startsymbol. $S \in V$.

Es gilt: $V \cap T = \emptyset$.

Betrachten wir beispielsweise eine Grammatik, die den Aufbau von Log-Einträgen beschreibt:

Beispiel 1.2.1.
$G = (\{\text{Start}, \text{Type}, \text{Text}, \text{ErrorCode}, \text{TimeStamp}\}, \{\texttt{INFO}, \texttt{WARNING}, \texttt{ERROR}, \texttt{:}, \texttt{.}, \texttt{ }, \texttt{--}, \texttt{(}, \texttt{)}, \text{NUMBER}, \text{STRING}\}, P, \text{Start})$
Wobei P aus folgenden Regeln besteht:

$$\begin{aligned}
\text{Start} &\to \text{TimeStamp } \texttt{--} \text{ Type } \texttt{:} \text{ Text } \texttt{(} \text{ ErrorCode } \texttt{)} \\
\text{Type} &\to \texttt{INFO} \mid \texttt{WARNING} \mid \texttt{ERROR} \\
\text{TimeStamp} &\to \text{NUMBER } \texttt{.} \text{ NUMBER } \texttt{.} \text{ NUMBER NUMBER } \texttt{:} \text{ NUMBER } \texttt{:} \text{ NUMBER} \\
\text{Text} &\to \text{STRING Text} \mid \text{NUMBER Text} \mid \epsilon \\
\text{ErrorCode} &\to \text{NUMBER} \mid \epsilon
\end{aligned}$$

Häufig wird die Grammatik nicht mehr als Tupel angegeben, sondern lediglich als Aufzählung ihrer

Produktionen. Dies ist dann möglich, wenn die Definition entsprechend beschränkt wird – dann ist das Startsymbol direkt ablesbar und ebenso auch die Mengen der Terminale und Variablen.

Kommen wir nun zu **regulären Grammatiken**.

Definition 1.2.2.
Eine **kontextfreie Grammatik** ist genau dann **regulär**, falls sie rechts-linear oder links-lineare ist. Rechts-linear ist eine Grammatik genau dann, wenn ihre Produktionen von der Form

$$A \to w\ B$$

oder

$$A \to\ w$$

und A, B Variablen sind und w eine Zeichenkette von Terminalen ist. Entsprechend definieren wir die Links-Linearität: Falls alle Produktionen der Grammatik der Form

$$A \to B\ w$$

oder

$$A \to w$$

sind, so ist die Grammatik links-linear.

Betrachten wir ein Beispiel für eine reguläre Grammatik:

Beispiel 1.2.2.
Die Sprache a(b)$*$ wird von der rechts-linearen Grammatik

$$S \to \text{a}\ A$$
$$A \to \text{b}\ A \mid \epsilon$$

und von der links-linearen Grammatik

$$S \to S\ \text{b} \mid \text{a}$$

erzeugt.

Bemerkung 2. Wir erkennen eine Rekursivität: In beiden Fällen gibt es mindestens eine Variable, die sich selbst in einer ihrer Produktionen enthält. Daher wird die Links-Linearität bzw. Rechts-Linearität gelegentlich auch **Links-Rekursion** oder **Rechts-Rekursion** genannt.

Nun erinnern wir uns an den Zusammenhang zwischen regulären Mengen und endlichen Automaten – gilt etwas Entsprechendes auch für reguläre Grammatiken?

Satz 1.2.1.
Reguläre Grammatiken und endliche Automaten sind äquivalent.

Wir geben an dieser Stelle nur eine Beweisidee an, der vollständige Beweis ist bei [1, S. 228] zu finden. Um die Aussage zu beweisen, nehmen wir an, $L = L(G)$ gilt für eine rechts-lineare Grammatik $G = (V, T, R, S)$. Dann können wir einen NEA mit ϵ-Übergangen aus den Ableitungsschritten konstruieren. Die Zustände dieses Automaten sind jeweils die linke oder rechte Seite der Produktionen. Nach Abbildung 1.2 (samt entsprechender Beweise) erhalten wir nun, dass die rechts-lineare Grammatik eine reguläre Menge erzeugt. Für eine links-lineare Grammatik gilt das Beweisprinzip analog.

1.3. Typ 2: Kontextfreie Grammatiken

Kontextfreie Grammatiken haben wir bereits im vorherigen Abschnitt definiert und genutzt, um reguläre Grammatiken zu definieren. Dies sparen wir uns also hier uns verweisen auf 1.2.1.

Wofür nutzen wir kontextfreie Grammatiken? Wie reguläre Grammatiken auch nutzen wir kontextfreie Grammatiken für die Definition und Übersetzung von Programmiersprachen. Eine wichtige Aufgabe, die durch reguläre Grammatiken nicht abgebildet werden kann, ist die korrekte Interpretation von geklammerten arithmetischen Ausdrücken (z.B. $(a + b) * c$) oder auch Code-Abschnitten wie `\begin{}` und `\end{}` in LaTeX. Über reguläre Sprachen ließen sich diese Strukturen nicht abbilden. [vgl. 1, S. 81]

1.3.1. Kellerautomaten

Nun besinnen wir uns nochmal auf Satz 1.2.1 zurück. Reguläre Grammatiken, also der Typ 3 der Chomsky-Hierarchie, sind äquivalent zu endlichen Automaten. Diese Eigenschaft ließ sich nutzen, um reguläre Grammatiken zu visualisieren und Automatenoperationen auf ihnen auszuführen. Mittlerweile befinden wir uns in einem anderen Chomsky-Typ – das legt nahe, dass kontextfreie Grammatiken auch einen Teil solcher Grammatiken umfassen, die nicht äquivalent zu endlichen Automaten sind. Doch gibt es ein Automatenkonzept für kontextfreie Grammatiken? Ja, die Kellerautomaten.

Informatikerbiotop Keller Vielleicht kommt Ihnen der Keller heimisch vor, vielleicht haben Sie dort (wie der Vater eines Schulfreundes) ihr Büro eingerichtet, von dem aus sie tagelang an einer Software herumprogrammieren, die Sie dann für viel Geld verkaufen. Vielleicht sind sie so beschäftigt, dass sich mit Essensresten verzierte Teller übereinander stapeln, weil Sie gutem Essen nie mehr Relevanz als einer Nebenbeitätigkeit beimessen. Behalten Sie diesen Tellerstapel – doch stellen Sie sich sie sauber vor, in einem ordentlichen Stapel auf einer Feder wie in einer Mensa-Schlange Ihrer Hochschule. Jeder Gast nimmt sich einen Teller vom Stapel, sobald der Stapel leer ist, werden neue Teller nachgelegt. Der letzte nachgelegte Teller ist der erste, der anschließend heruntergenommen wird – wir folgern: Ein Last-in-First-out-Prinzip (LIFO). Nun spätestens sollte Ihnen sofort die dazugehörige Datenstruktur einfallen, falls Sie den Begriff nicht sowieso direkt mit „Keller“ assoziiert haben: **Stack**. Wir ergänzen endliche Automaten um einen Stack, der nun mit beeinflusst, welche Zustandsübergänge erfolgen.

Definition 1.3.1.
Ein Kellerautomat M ist ein System $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, wobei ...

- Q eine endliche Menge von Zuständen
- Σ ein Eingabealphabet
- Γ ein Kelleralphabet
- $q_0 \in Q$ ein Anfangszustand
- $Z_0 \in \Gamma$ ein Anfangssymbol
- $F \subseteq Q$ eine Menge von Endzuständen
- δ eine Abbildung der Form $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \mapsto \mathcal{P}(Q \times \Gamma*)$

ist.

Die Ähnlichkeit zur Definition endlicher Automaten ist augenscheinlich. Hinzu kommen ein Kelleralphabet (Worte, die auf den Stack gelegt werden), ein Anfangssymbol, dass sich zuallererst auf dem Stack befindet und eine veränderte Zustandsübergangsfunktion, die nun auch das oberste Element des Stacks mitberücksichtigt.

$$\delta(q, a, Z) = \{(p_i, \gamma_i), ..., (p_m, \gamma_m)\}$$

Hier bezeichnet q den aktuellen Zustand, a das nächste eingelesene Element von Σ und Z das oberste Kellersymbol. p_i ist Element der Zustandsmenge Q und γ_i eine Folge von Symbolen des Kelleralphabets. Grundsätzlich existieren drei Stack-Operationen:

- **Hinzufügen eines Elements**. Liegt vor der Operation ein X auf dem Stack und soll ein Y hinzugefügt werden, so ist der Stack anschließend YX.

- **Entfernen eines Elements.** $XY \to Y$
- **Unverändertlassen.**

1.4. Typ 1: Kontextsensitive Grammatiken

1.5. Typ 0: Rekursiv aufzählbare Grammatiken

Literatur

[1] J. E. Hopcroft und J. D. Ullman, *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 1988, ISBN: 3-925118-82-9.

[2] M. Sipser, *Introduction to the Theory of Computation*, 3. Aufl. Cengage Learning, 2013, ISBN: 978-1-133-18781-3.

[3] M. Ben-Ari, *Mathematical Logic for Computer Science*, 2. Aufl. London: Springer-Verlag, 2001, ISBN: 1-85233-319-7.