TECHNISCHE UNIVERSITÄT DRESDEN

FORMALE SYSTEME

1. Vorlesung: Willkommen/Einleitung formale Sprachen

Markus Krötzsch

Lehrstuhl Wissensbasierte Systeme

TU Dresden, 9. Oktober 2017

Willkommen zur Vorlesung Formale Systeme

Raum, Zeit, URL

- **Vorlesungen:**
  Montag, DS3 (11:10–12:40), HSZ/0002
  Donnerstag, DS4 (13:00–14:30), HSZ/0003
- **Keine Vorlesungen:**
  Mo 23.10.
  Jahreswechsel: Do 21.12., Mo 25.12., Do 28.12., Mo 1.1.
- **Vorlesungswebseite**:
  `https://iccl.inf.tu-dresden.de/web/FS2017`
  (Folien, Übungsblätter, Termine, etc.)
- **Quellen, Bug-Reports, Fragen**:
  `https://github.com/mkroetzsch/FormaleSysteme`
  (Pull-Requests sind willkommen)
- **Mailingliste**:
  `https://mailman.zih.tu-dresden.de/groups/listinfo/inf-thi-fs1718`
  (Support bei allen Fragen zur Vorlesung)

Übungen

- **Anmeldung** zu den Übungen über jExam
- **Übungsblätter** jeweils donnerstags nach der Vorlesung (erstes Übungsblatt am 12. Oktober 2017)
- **Beginn der Übungen:** 16. Oktober 2017
- **Übungsablauf**, vereinfacht, idealisiert:
  Aufgaben werden zu Hause bearbeitet so gut es geht;
  in der Übung helfen Gruppenleiter/innen bei Fragen und Problemen und zeigen Beispiellösungen

Prüfung und Prüfungsvorbereitung

- schriftliche Prüfung (90min) am Ende des Wintersemesters
- prüfungsrelevant:
  kompletter Stoff, der in der Vorlesung behandelt wird
  Wiedergeben (Definieren) und Anwenden (Rechnen)
- Modulnote ergibt sich je nach Studiengang
- zur zusätzlichen Vorbereitung gibt es 2–3 Repetitorien und eine Probeklausur, jeweils an einem Vorlesungstermin

Formale Systeme Bestehen

Tipps:

- **Von Hand Mitschreiben**
  Man merkt sich Stoff deutlich besser, wenn man ihn für sich selbst handschriftlich zusammenfasst.[1]
- **Selber Rechnen**
  Die Prüfung besteht im Lösen von Rechenaufgaben. Theorie allein hilft da nicht weiter.
- **Schnell sein**
  Prüfungszeit ist meistens knapp. Es reicht nicht, Aufgaben „im Prinzip“ lösen zu können. Man muss sie schnell lösen.
- **Ehrlich zu sich selbst sein**
  Man sollte selbst wissen, ob man genug gelernt hat oder nicht.[2]

[1] P. Mueller & D. Oppenheimer. The Pen Is Mightier Than the Keyboard: Advantages of Longhand Over Laptop Note Taking. Psychological Science, 06/2014, 25:6

[2] Vgl. aber auch Wikipedia [[Dunning-Kruger-Effekt]]

Übersicht und Motivation

Grundlegende Fragen der Informatik

Was ist ein Computer?

Eine Maschine, die rechnet.

Was ist „Rechnen“?

Die systematische Überführung von Eingaben in Ausgaben.

Was sind „Eingaben“ und „Ausgaben“?

Folgen von Zeichen, zum Beispiel Dateien oder Textausgaben.

So viele Computer, Programme, . . . das passt doch in kein Studium!

Nein, man muss sich auf das Wesentliche konzentrieren.

Was ist das Wesentliche?

Vereinfachte Modelle für Computer und Rechenverfahren.

Was sagen uns Modelle über echte Computer und Software?

Was man berechnen kann, wie aufwändig es ist,
wie man es implementieren kann, ob es stimmt, . . .

Zielstellung, Kernthemen

Ziel dieser Vorlesung ist es, wichtige Grundlagen zur Modellierung von Berechnung in der Informatik einzuführen, konkrete Modelle vorzustellen und ihre Eigenschaften verständlich zu machen.

- Ein- und Ausgaben sind Zeichenfolgen
  ⇝ Wir beginnen mit Wörtern und formalen Sprachen
- Wir wollen Berechnungsaufgaben beschreiben
  ⇝ Spezifikation von Sprachen
  (direkt, mit Grammatiken, mit regulären Ausdrucken, . . .)
- Fokus auf einfache Berechnungsmodelle
  ⇝ Automaten (in vielen Versionen . . .)
- Man kann Berechnungsaufgaben auch logisch spezifizieren
  ⇝ Aussagenlogik als einfacher Einstieg
- Lösung logischer Probleme
  ⇝ Berechnungsverfahren zum logischen Schließen

Gliederung „Formale Systeme“

Teil 1: Sprachen und Automaten

- Formale Sprachen und Grammatiken
- Reguläre Sprachen und endliche Automaten
- Kontextfreie Sprachen und Kellerautomaten
- Kontextsensitive Sprachen, Typ-0-Sprachen und Turingmaschinen

Teil 2: Aussagenlogik

- Syntax und Semantik
- logisches Schließen, Backtracking und andere Verfahren
- Horn-Logik als Vereinfachung

Literatur: Lehrbücher

Der Vorlesungsstoff gehört zu fast jeder Informatikausbildung. Es gibt viele Lehrmaterialien und eine weitgehend einheitliche Notation.

Lehrbücher zum ersten Teil der Vorlesung (formale Sprachen):

- Uwe Schöning: **Theoretische Informatik – kurz gefasst.** Spektrum Akademischer Verlag
  deutschsprachiger Standardtext; in der Tat ziemlich kurz gefasst
- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman: **Einführung in Automatentheorie, Formale Sprachen und Berechenbarkeit.** Pearson Studium
  aus dem Englischen übertragenes Standardwerk; Original ev. besser
- Michael Sipser: **Introduction to the Theory of Computation.** Cengage Learning
  Standardtext zu Sprachen und Berechnungskomplexität; leider nur auf Englisch

Literatur: Frei zugängliche Skripte

Es gibt mehrere Skripte zu dieser Vorlesung:

- Franz Baader: **Skript Formale Systeme, Teil 1 – Automaten und formale Sprachen.** TU Dresden.
  Siehe https://lat.inf.tu-dresden.de/teaching/ws2013-2014/FS/script_2016-02.pdf.
- Christel Baier, Manuela Berg, Walter Nauber: **Formale Systeme WS 2011/2012: Skript zur Vorlesung.** TU Dresden.
  Siehe http://www.inf.tu-dresden.de/content/institutes/thi/algi/lehre/WS1415/FS/lecture_notes/Skript_FS_C_Baier_WS1112.pdf.

Die Vorlesungen haben kleine Abweichungen, stimmen aber in vielen wichtigen Punkten überein.

Sprachen in der Informatik

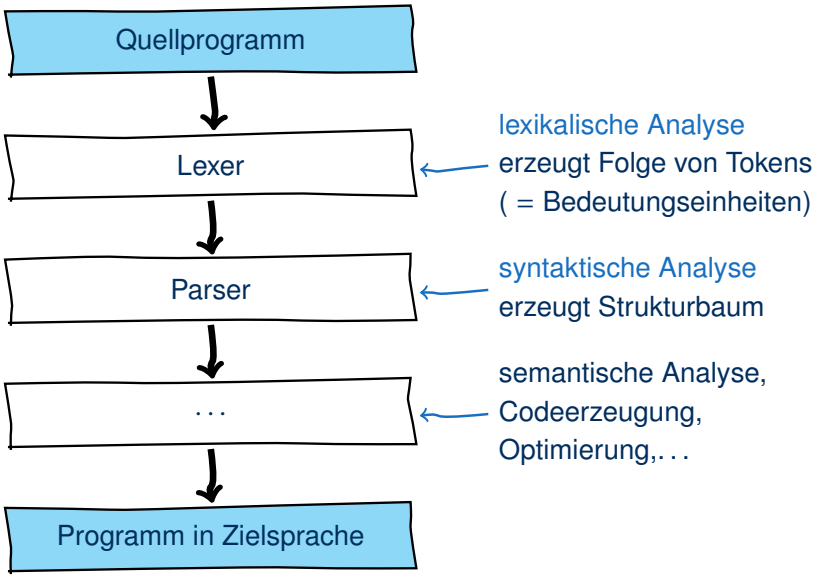
Randall Munroe, http://xkcd.com/208/, CC-BY-NC 2.5

Wozu Sprachen?

Formale Sprachen und die zugehörigen Automaten und Grammatiken haben sehr viele Anwendungen:

- Compilerbau
  Programmiersprachen sind typische formale Sprachen
- Interpretation natürlicher Sprachen
  viele Anwendungen des Sprachverstehens nutzen Grammatiken
- Datenaustausch
  Textformate (HTML, CSV, JSON, XML, . . .) bilden formale Sprachen
- Formatierung/Validierung/Spezifikation
  z.B. um die Gültigkeit von Formulareingaben zu prüfen
- Berechenbarkeit und Komplexität
  mächtigere Sprachdefinitionen verlangen teurere Algorithmen
- Informationsextraktion
  Formale Sprachen helfen bei der Mustersuche in Textdokumenten
- Datenbanken
  Datenbankanfragen können Muster suchen, z.B. in Graphdatenbanken

Beispiel Compiler



Lexikalische Analyse

Eingabe: Zeichenkette eines Programms

Bsp.: `lengthCm = lengthInch * 2.54;`
Kette von 29 Zeichen

Ausgabe: Kette von Grundsymbolen (Tokens)

Bsp.: NAME EQUALS NAME STAR NUMBER SEMICOLON
Kette von 6 Tokens

- Zur Weiterverarbeitung wird Tokens oft weitere Information mitgegeben, z.B. NAME("`lengthCm`") und NAME("`lengthInch`")
- Manche Zeichen werden nicht zu Tokens (z.B. Leerzeichen, Kommentare)

Lexikalische Analyse (2)

Verschiedene Arten von Grundsymbolen:

- Schlüsselwörter (`if`, `while`, `class`, ...)
- Operatoren (`=`, `+`, `>>`, ...)
- Bezeichner (`lengthCm`, `getLength`, `InventoryItem`, ...)
- Literale (`2.54`, `true`, `"HelloWorld!"`, ...)

Für einige Grundsymbole gibt es unendlich viele Möglichkeiten

Bsp.: „Ein Bezeichner ist ein String, der mit einem Buchstaben beginnt und danach nur Buchstaben oder Ziffern enthält.“

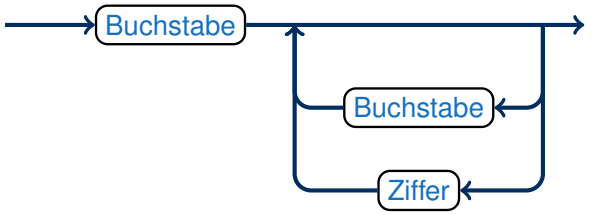
Wie soll ein Lexer das korrekt erkennen?

Bezeichner erkennen

Bsp.: „Ein Bezeichner ist ein String, der mit einem Buchstaben beginnt und danach nur Buchstaben oder Ziffern enthält.“

Schematische Darstellung als Syntaxdiagramm:

Bezeichner:



Hierbei stehen Buchstabe und Ziffer jeweils für ein beliebiges Zeichen dieses Typs.

Bezeichner erkennen (2)

Wie setzt man das praktisch um?
Code eines unerfahrenen Programmierers:

```
function isIdentifier():
  state = "start"
  while hasNextSymbol():
     symbol = getNextSymbol()
     if ( state == "start" && isLetter(symbol) ):
        state = "inner"
     else if ( state == "start" && !isLetter(symbol) ):
        return false
     else if ( state == "inner" && isLetter(symbol) ):
        // ok, wir lesen einfach weiter
     else if ( state == "inner" && isNumber(symbol) ):
        // ok, wir lesen einfach weiter
     else if ( state == "inner" &&
               !isLetter(symbol) && !isNumber(symbol) ):
        return false
  if (state == "inner"): return true
```

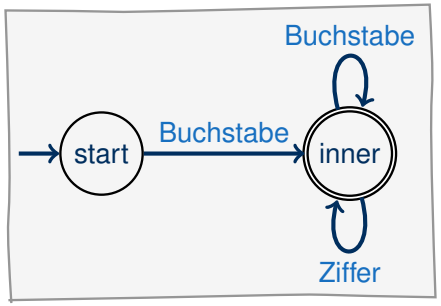

Bezeichner erkennen (3)

Der (schlechte) Programmcode zeigt eine wichtige Eigenschaft:

Der Lexer muss nur einen „Zustand“ speichern

(im Beispiel ist dies der Wert der Variable `state`)

Darstellung als endlicher Automat:



- Automat beginnt im Startzustand → start
- Zustandswechsel gemäß Pfeilen
- Kein passender Pfeil für Symbol?
  „`return false`“
- Keine weiteren Symbole?
  „`return true`“ in Endzustand inner
  „`return false`“ falls in Zustand start

Bezeichner erkennen (4)

Wie kann man jemandem am besten erklären, was ein „Bezeichner“ ist?

- Sprachliche Umschreibung: ungenau und mehrdeutig
- Syntaxdiagram, endlicher Automat: graphische Darstellung anschaulich, aber schnell unübersichtlich
- Programmcode: Kernidee geht in Implementierungsdetails verloren

⇝ Spezifikationen verwenden meist Grammatiken

Bezeichner ::= Buchstabe | Buchstabe InBezeichner
InBezeichner ::= BuchOderZiff | BuchOderZiff InBezeichner
BuchOderZiff ::= Buchstabe | Ziffer
Buchstabe ::= "a" | "b" | ... | "Z"
Ziffer ::= "0" | "1" | ... | "9"

Formale Sprachen

Wir haben gesehen:

- In der Praxis interessieren wir uns für (oftmals unendliche) Mengen von Strings, z.B. Bezeichner oder Java Programme
  ⇝ solche Mengen nennt man formale Sprachen
- Man kann formale Sprachen auf viele Arten beschreiben, z.B. mit Automaten, Grammatiken oder Syntaxdiagrammen
  ⇝ unterschiedliche Stärken und Schwächen
- Kein Ansatz kann alle Sprachen beschreiben, z.B. gibt es keinen Automaten, der Java-Programme parsen kann
  ⇝ man unterscheidet Typen von Sprachen

Erster Teil der Vorlesung:
formale Sprachen und ihre Darstellungsarten

Formale Sprachen

Grundbegriffe: Alphabete und Wörter

Ein Alphabet ist eine endliche, nicht-leere Menge.
Elemente des Alphabets heißen Symbole.

Meistens verwendet man die Buchstaben Σ (Sigma) oder Γ (Gamma) für Alphabete.

Ein endliches Wort über einem Alphabet Σ ist eine endliche Folge von Symbolen aus Σ. Wenn w ein endliches Wort ist, dann ist $|w|$ seine Länge (die Anzahl seiner Symbole).

Alle Wörter in dieser Vorlesung sind endlich. Wir sagen das ab jetzt nicht immer dazu.

Beispiel: $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ ist ein Alphabet.
Wörter über diesem Alphabet sind z.B. $\mathtt{abba}$ oder $\mathtt{bbb}$.
Die Längen dieser Wörter sind $|\mathtt{abba}| = 4$ und $|\mathtt{bbb}| = 3$.

Grundbegriffe: Konkatenation, Leeres Wort

Die wichtigste Operation auf Wörtern ist Konkatenation („Hintereinanderhängung“):

Die Konkatenation von zwei Wörtern $w = \mathtt{a}_1 \dots \mathtt{a}_n$ und $v = \mathtt{b}_1 \dots \mathtt{b}_m$ ist das Wort $wv = \mathtt{a}_1 \dots \mathtt{a}_n \mathtt{b}_1 \dots \mathtt{b}_m$.

Wir schreiben also konkatenierte Wörter einfach nebeneinander.

Das leere Wort ϵ (epsilon) ist das Wort der Länge 0, also $|\epsilon| = 0$.

Es gibt genau ein leeres Wort und man kann es über jedem Alphabet bilden.

Beispiel: Für die Wörter $w = \mathtt{tuben}$ und $v = \mathtt{wachs}$ gilt $wv = \mathtt{tubenwachs}$ und $vw = \mathtt{wachstuben}$.

Beispiel: Für jedes Wort w gilt: $w\epsilon = \epsilon w = w$

Grundbegriffe: *-fixe

Manchmal ist es praktisch, Teile von Wörtern zu bezeichnen:

Sei $w = \mathtt{a}_1 \dots \mathtt{a}_n$ ein Wort der Länge n.

- Ein Präfix von w ist ein Wort $\mathtt{a}_1 \dots \mathtt{a}_i$ mit $0 \leq i \leq n$
- Ein Suffix von w ist ein Wort $\mathtt{a}_j \dots \mathtt{a}_n$ mit $0 \leq j \leq n$
- Ein Infix von w ist ein Wort $\mathtt{a}_i \dots \mathtt{a}_j$ mit $0 \leq i \leq j \leq n$

Beispiel: Das Wort $\mathtt{Staubecken}$ hat ein Präfix $\mathtt{Staub}$, ein Suffix $\mathtt{ecken}$ und ein Infix $\mathtt{taube}$ (und viele andere mehr).

Beispiel: Das leere Wort ϵ ist Präfix, Suffix und Infix von jedem Wort (sogar von ϵ selbst).

Grundbegriffe: formale Sprache

Sei Σ ein Alphabet. Eine Menge von Wörtern über Σ wird formale Sprache über Σ genannt.

Die Zusätze „formal“ und „über Σ“ werden meist weggelassen, wenn dadurch keine Missverständnisse auftreten können.

Sprachen werden meist mit dem Buchstaben **L** bezeichnet.

Beispiel: Die Sprache **Ziffer** $= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ ist eine endliche Sprache (über jedem Alphabet, das zumindest die Symbole 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 enthält).

Beispiel: Die zuvor genannte Sprache **Bezeichner** ist unendlich.

Beispiel: Die Menge aller Tweets ist eine endliche Sprache.

Beispiel: Die Menge aller validen HTML-5-Dokumente ist eine unendliche Sprache über dem Alphabet aller Unicode Code Points.

Die kleinste und die größte Sprache

Die kleinste Sprache über dem Alphabet Σ ist die leere Sprache $\emptyset$.

Beispiel: Die Sprache $\{\epsilon\}$, welche nur das leere Wort enthält, ist nicht leer!

Die leere Sprache ist also Teilmenge jeder anderen Sprache.

Die größte Sprache über einem Alphabet Σ ist die Sprache aller Wörter über Σ. Man bezeichnet sie mit Σ^*.

Beispiel: $\{a, b\}^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$.

Jede Sprache über Σ ist also eine Teilmenge von Σ^*.

Operationen auf Sprachen

Man kann mit Sprachen „rechnen“ um neue Sprachen zu bilden:

- Vereinigung: $\mathbf{L_1} \cup \mathbf{L_2}$
- Schnitt: $\mathbf{L_1} \cap \mathbf{L_2}$
- Komplement: $\overline{\mathbf{L}} = \Sigma^* \setminus \mathbf{L}$ (nur sinnvoll, wenn das Alphabet Σ klar festgelegt ist!)
- Produkt: $\mathbf{L_1} \circ \mathbf{L_2} = \{w_1 w_2 \mid w_1 \in \mathbf{L_1}, w_2 \in \mathbf{L_2}\}$
- Potenz: $\mathbf{L}^0 = \{\epsilon\}$ und $\mathbf{L}^{n+1} = \mathbf{L} \circ \mathbf{L}^n$
- Kleene-Abschluss: $\mathbf{L}^* = \mathbf{L}^0 \cup \mathbf{L}^1 \cup \mathbf{L}^2 \cup \ldots = \bigcup_{i \geq 0} \mathbf{L}^i$

Beispiel: Die Sprache **Bezeichner** kann aus den Sprachen **Buchstabe** und **Ziffer** konstruiert werden:

$$\textbf{Bezeichner} = \textbf{Buchstabe} \circ (\textbf{Buchstabe} \cup \textbf{Ziffer})^*$$

Beispiel: Tweets sind Ketten aus bis zu 140 Zeichen:[1]

$$\textbf{Tweet} = \bigcup_{1 \leq i \leq 140} \textbf{Buchstabe}^i$$

[1] korrekt bis etwa Mai 2016

Operationen auf Sprachen: Kurzschreibweisen

- Der Produktoperator wird oft eingespart:
  statt $\mathbf{L_1} \circ \mathbf{L_2}$ schreiben wir $\mathbf{L_1 L_2}$
  Hinweis: Manche Autoren verwenden $\cdot$ statt $\circ$.
- Ein Plus-Operator ist manchmal praktisch: $\mathbf{L}^+ = \bigcup_{i \geq 1} \mathbf{L}^i$
- Die Differenz ist indirekt ausdrückbar: $\mathbf{L_1} \setminus \mathbf{L_2} = \mathbf{L_1} \cap \overline{\mathbf{L_2}}$
- Klammern einsparen: Kleene-Stern * und Plus $^+$ binden immer am stärksten, gefolgt von $\circ$, $\cap$, $\cup$ und $\setminus$

Manchmal sind Klammern und $\circ$ aber zur Lesbarkeit trotzdem hilfreich!

Beispiel: Die Menge aller gültigen Schreibweisen für Dezimalzahlen kann wie folgt definiert werden:

$$\textbf{Dezimalzahl} = \Big(\{\epsilon\} \cup \{+, -\}\Big) \circ \Big(\{0\} \cup (\mathbf{Z} \setminus \{0\}) \circ \mathbf{Z}^*\Big) \circ \Big(\{\epsilon\} \cup \{.\}\mathbf{Z}^+\Big)$$

mit $\mathbf{Z} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Ausdrücke der Form $(\{\epsilon\} \cup \mathbf{L})$ beschreiben optionale Bestandteile.

Rechenregeln (1)

Es gelten viele Rechenregeln für Operationen auf Sprachen.

Sprachen sind Mengen – es gelten die Gesetze der Mengenlehre.

Für Sprachen $\mathbf{L_1}$, $\mathbf{L_2}$, $\mathbf{K}$ über einem gemeinsamen Alphabet Σ gilt:

- $\mathbf{L_1} \cap \mathbf{L_2} = \mathbf{L_2} \cap \mathbf{L_1}$ (Kommutativgesetz $\cap$)
- $\mathbf{L_1} \cup \mathbf{L_2} = \mathbf{L_2} \cup \mathbf{L_1}$ (Kommutativgesetz $\cup$)
- $\mathbf{K} \cup (\mathbf{L_1} \cap \mathbf{L_2}) = (\mathbf{K} \cup \mathbf{L_1}) \cap (\mathbf{K} \cup \mathbf{L_2})$ (Distributivgesetz $\cup$ über $\cap$)
- $\mathbf{K} \cap (\mathbf{L_1} \cup \mathbf{L_2}) = (\mathbf{K} \cap \mathbf{L_1}) \cup (\mathbf{K} \cap \mathbf{L_2})$ (Distributivgesetz $\cap$ über $\cup$)
- $\mathbf{L_1} \cup \mathbf{L_2} = \overline{\overline{\mathbf{L_1}} \cap \overline{\mathbf{L_2}}}$ (De Morgansches Gesetz 1)
- $\mathbf{L_1} \cap \mathbf{L_2} = \overline{\overline{\mathbf{L_1}} \cup \overline{\mathbf{L_2}}}$ (De Morgansches Gesetz 2)

Rechenregeln (2)

Es gelten weitere Gesetze für die zusätzlichen Sprachoperationen.

Für Sprachen $\mathbf{L_1}$, $\mathbf{L_2}$, $\mathbf{K}$ über einem gemeinsamen Alphabet Σ gilt:

- $\mathbf{K} \circ (\mathbf{L_1} \cup \mathbf{L_2}) = (\mathbf{K} \circ \mathbf{L_1}) \cup (\mathbf{K} \circ \mathbf{L_2})$
  $(\mathbf{L_1} \cup \mathbf{L_2}) \circ \mathbf{K} = (\mathbf{L_1} \circ \mathbf{K}) \cup (\mathbf{L_2} \circ \mathbf{K})$
  } Distributivgesetze $\circ/\cup$
- Achtung: Distributivgesetze $\circ/\cap$ gelten nicht!
- $\mathbf{K} = \mathbf{K} \circ \{\epsilon\} = \{\epsilon\} \circ \mathbf{K}$ ($\{\epsilon\}$ neutrales Element zu $\circ$)
- $\mathbf{K} \circ \emptyset = \emptyset \circ \mathbf{K} = \emptyset$ ($\emptyset$ absorbierendes Element zu $\circ$)
- $\mathbf{K}^+ = \mathbf{K}^* \circ \mathbf{K} = \mathbf{K} \circ \mathbf{K}^*$
- $\mathbf{K}^* = \mathbf{K}^+ \cup \{\epsilon\} = (\mathbf{K} \setminus \{\epsilon\})^*$

Rechenregeln beweisen?

Behauptung: $\mathbf{K} \circ (\mathbf{L_1} \cup \mathbf{L_2}) = (\mathbf{K} \circ \mathbf{L_1}) \cup (\mathbf{K} \circ \mathbf{L_2})$

Beweis: Wir schauen uns einfach die Definitionen der jeweiligen Operationen an.

$$\begin{aligned}
\mathbf{K} \circ (\mathbf{L_1} \cup \mathbf{L_2}) &= \{wv \mid w \in \mathbf{K};\ v \in (\mathbf{L_1} \cup \mathbf{L_2})\} \\
&= \{wv \mid w \in \mathbf{K};\ v \in \mathbf{L_1} \text{ oder } v \in \mathbf{L_2}\} \\
&= \{wv \mid w \in \mathbf{K};\ v \in \mathbf{L_1}\} \cup \{wv \mid w \in \mathbf{K};\ v \in \mathbf{L_2}\} \\
&= (\mathbf{K} \circ \mathbf{L_1}) \cup (\mathbf{K} \circ \mathbf{L_2})
\end{aligned}$$

(Allgemein gilt: jede Operation, die „punktweise“ auf einer Menge arbeitet, distribuiert über $\cup$) □

Zusammenfassung und Ausblick

Formale Sprachen sind für die Informatik sehr wichtig

Grundbegriffe: Alphabet, Wort, Sprache, Konkatenation, Präfix/Suffix/Infix, ϵ, Σ^*

Es gibt eine Reihe von Operationen auf Sprachen, mit denen man rechnen kann

Offene Fragen:

- Wie viele Wörter und wie viele Sprachen gibt es eigentlich?
- Wie kann man Sprachen beschreiben oder implementieren?
- Wie kommt Berechnung ins Spiel?