Theoretische Informatik

Matthias Heizmann

Software Engineering
Albert-Ludwigs-University Freiburg

October 18, 2017

- Motivation / Vorlesungsinhalt
- Organisation
- Erstes Kapitel: Formale Sprachen

Motivation

Richtig oder falsch?

"Computer können alles berechnen, wir brauchen nur genug Rechenleistung und genug Arbeitsspeicher."

Foto: SuperMUC am Leibniz-Rechenzentrum

Richtig oder falsch?

"Manche Probleme kann man nur mit bestimmten Programmiersprachen lösen."

Richtig oder falsch?

"Wenn man sich genug anstrengt kann man alle Computerprogramme verstehen."

Lernziele: Methodik

- Abstrakte Betrachtung von Computern und Berechnungsverfahren

Lernziele: Methodik

- Abstrakte Betrachtung von Computern und Berechnungsverfahren
- Betrachtung von Computern und Berechnungsverfahren mit mathematischen Formalismen

Warum abstrakte Betrachtung?

- Konkrete Hardware/Software sehr komplex

Warum abstrakte Betrachtung?

- Konkrete Hardware/Software sehr komplex
  - Bsp. CPU: Processor Programming Reference (PPR) for AMD Family 17h Model 01h, Revision B1 Processor hat ca. 250 Seiten.
  - Bsp. Programmiersprache: ISO/IEC 9899:2011 ("C11 Standard") hat ca. 700 Seiten.

Warum abstrakte Betrachtung?

- Konkrete Hardware/Software sehr komplex
  - Bsp. CPU: Processor Programming Reference (PPR) for AMD Family 17h Model 01h, Revision B1 Processor hat ca. 250 Seiten.
  - Bsp. Programmiersprache: ISO/IEC 9899:2011 ("C11 Standard") hat ca. 700 Seiten.
- Wachsende Vielfalt von Hardware und Programmiersprachen

Warum abstrakte Betrachtung?

- Konkrete Hardware/Software sehr komplex
  - Bsp. CPU: Processor Programming Reference (PPR) for AMD Family 17h Model 01h, Revision B1 Processor hat ca. 250 Seiten.
  - Bsp. Programmiersprache: ISO/IEC 9899:2011 ("C11 Standard") hat ca. 700 Seiten.
- Wachsende Vielfalt von Hardware und Programmiersprachen
  - 70er Jahre: Basic, Cobol, Fortran, PL/I, ...
  - Heute: C#, Haskell, Java, Python, ...
  - 2040: ???

Warum mathematische Formalismen?

Missverständnisfreie Kommunikation ermöglichen!

Warum mathematische Formalismen?

Missverständnisfreie Kommunikation ermöglichen!

1. Es kann keinen Compiler geben, der immer herausfindet ob das Eingabeprogramm immer terminiert.

Warum mathematische Formalismen?

Missverständnisfreie Kommunikation ermöglichen!

1. Es kann keinen Compiler geben, der immer herausfindet ob das Eingabeprogramm immer terminiert.
2. Für jedes $n \in \mathbb{N}$ gilt: Für jedes Computerprogramm, das höchstens n Byte Speicher benötigt, können wir entscheiden ob dieses terminiert.

Widerspruch?

Warum mathematische Formalismen?

Missverständnisfreie Kommunikation ermöglichen!

1. Es kann keinen Compiler geben, der immer herausfindet ob das Eingabeprogramm immer terminiert.
2. Für jedes $n \in \mathbb{N}$ gilt: Für jedes Computerprogramm, das höchstens n Byte Speicher benötigt, können wir entscheiden ob dieses terminiert.

Widerspruch?

- Was ist ein Programm?
- Was bedeutet "Terminierung"?
- Was ist eine Eingabe?
- Was bedeutet "Speicher benötigen"?
- Was bedeutet "immer herausfinden"?

Lernziele: Inhalt

1. Formale Sprachen und Automatentheorie

Lernziele: Inhalt

1. Formale Sprachen und Automatentheorie
   "Rechnen mit Zeichenketten und Mengen von Zeichenketten"

Lernziele: Inhalt

1. Formale Sprachen und Automatentheorie
   "Rechnen mit Zeichenketten und Mengen von Zeichenketten"
   - Endliche Automaten
   - Reguläre Ausdrücke
   - Grammatiken
   - Automaten mit Stapelspeicher
   - Turingmaschinen

Lernziele: Inhalt

1. Formale Sprachen und Automatentheorie
   "Rechnen mit Zeichenketten und Mengen von Zeichenketten"
   - Endliche Automaten
   - Reguläre Ausdrücke
   - Grammatiken
   - Automaten mit Stapelspeicher
   - Turingmaschinen
2. Berechenbarkeitstheorie
   - Nicht berechenbare Funktionen
   - Unentscheidbare Probleme
   - Reduktion

Lernziele: Inhalt

1. Formale Sprachen und Automatentheorie
   "Rechnen mit Zeichenketten und Mengen von Zeichenketten"
   - Endliche Automaten
   - Reguläre Ausdrücke
   - Grammatiken
   - Automaten mit Stapelspeicher
   - Turingmaschinen
2. Berechenbarkeitstheorie
   - Nicht berechenbare Funktionen
   - Unentscheidbare Probleme
   - Reduktion
3. Komplexitätstheorie
   - Berechnungskomplexität auf Turingmaschinen
   - Komplexitätsklassen P und NP

Organisation

siehe Vorlesungswebsite

Vorspann: Formale Sprachen

Grundbegriffe, Notationen

Alphabet Σ

Ein *Alphabet* ist eine endliche Menge von *Zeichen*.

Zeichen sind hier beliebige abstrakte Symbole.

Alphabet Σ

Ein *Alphabet* ist eine endliche Menge von *Zeichen*.

Zeichen sind hier beliebige abstrakte Symbole.

Bsp

für Alphabete, die in dieser Vorlesung, im täglichen Umgang mit Computern oder in der Forschung an unserem Lehrstuhl eine Rolle spielen

- $\{a, \ldots, z\}$
- $\{0, 1\}$
- $\{rot, gelb, grün\}$ (Ampelfarben)
- Die Menge aller ASCII-Symbole
- Die Menge aller Statements eines Computerprogramms

Wort w über Σ

Wir nennen eine endliche Folge von Elementen aus Σ ein *Wort*.

Wir schreiben solche Folge ohne Trennsymbole.

z.B. `einhorn` statt `e,i,n,h,o,r,n`

Wort w über Σ

Wir nennen eine endliche Folge von Elementen aus Σ ein *Wort*.

Wir schreiben solche Folge ohne Trennsymbole.

z.B. `einhorn` statt `e,i,n,h,o,r,n`

Leeres Wort ε

Die leere Folge nennen wir das *leere Wort*.

Wort w über Σ

Wir nennen eine endliche Folge von Elementen aus Σ ein *Wort*.

Wir schreiben solche Folge ohne Trennsymbole.

z.B. `einhorn` statt `e,i,n,h,o,r,n`

Leeres Wort ε

Die leere Folge nennen wir das *leere Wort*.

Notation:

- Menge aller Wörter: Σ^*
- Menge aller nicht leeren Wörter: Σ^+

Wort w über Σ

Wir nennen eine endliche Folge von Elementen aus Σ ein *Wort*.

Wir schreiben solche Folge ohne Trennsymbole.

z.B. `einhorn` statt `e,i,n,h,o,r,n`

Leeres Wort ε

Die leere Folge nennen wir das *leere Wort*.

Notation:

- Menge aller Wörter: Σ^*
- Menge aller nicht leeren Wörter: Σ^+

Länge

Die *Länge* eines Wortes, $|\cdot| : \Sigma^* \to \mathbb{N}$, ist die Anzahl der Elemente der Folge.

Konkatenation von Wörtern

Die *Konkatenation*, $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$, ist für $u = u_1 \dots u_n \in \Sigma^*$ und $v = v_1 \dots v_m \in \Sigma^*$ definiert durch: $u \cdot v = u_1 \dots u_n v_1 \dots v_m$

Konkatenation von Wörtern

Die *Konkatenation*, $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$, ist für $u = u_1 \dots u_n \in \Sigma^*$ und $v = v_1 \dots v_m \in \Sigma^*$ definiert durch: $u \cdot v = u_1 \dots u_n v_1 \dots v_m$

- assoziativ
- nicht kommutativ!
- wir dürfen Klammern weglassen
- wir dürfen Operationssymbol weglassen

Konkatenation von Wörtern

Die *Konkatenation*, $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$, ist für $u = u_1 \dots u_n \in \Sigma^*$ und $v = v_1 \dots v_m \in \Sigma^*$ definiert durch: $u \cdot v = u_1 \dots u_n v_1 \dots v_m$

- assoziativ
- nicht kommutativ!
- wir dürfen Klammern weglassen
- wir dürfen Operationssymbol weglassen

Definition

Die *Potenzierung* von Wörtern, $\cdot^{\cdot} : \Sigma^* \times \mathbb{N} \to \Sigma^*$, ist induktiv definiert durch

1. $w^0 = \varepsilon$
2. $w^{n+1} = w \cdot w^n$

Sprache über Σ

Eine *Sprache* über Σ ist eine Menge $L \subseteq \Sigma^*$.

Sprache über Σ

Eine *Sprache* über Σ ist eine Menge $L \subseteq \Sigma^*$.

Konkatenation und Potenzierung von Sprachen

Seien $U, V \subseteq \Sigma^*$. Dann ist die *Konkatenation* von U und V definiert durch

$$U \cdot V = \{u \cdot v \mid u \in U, v \in V\}$$

und die *Potenzierung* von U induktiv definiert durch

1. $U^0 = \{\varepsilon\}$
2. $U^{n+1} = U \cdot U^n$

Sprache über Σ

Eine *Sprache* über Σ ist eine Menge $L \subseteq \Sigma^*$.

Konkatenation und Potenzierung von Sprachen

Seien $U, V \subseteq \Sigma^*$. Dann ist die *Konkatenation* von U und V definiert durch

$$U \cdot V = \{u \cdot v \mid u \in U, v \in V\}$$

und die *Potenzierung* von U induktiv definiert durch

1. $U^0 = \{\varepsilon\}$
2. $U^{n+1} = U \cdot U^n$

Kleene-Abschluss, Kleene-Stern

Sei $U \subseteq \Sigma^*$. Der *Kleene-Abschluss* ist definiert durch

1. $U^* = \bigcup_{n \in \mathbb{N}} U^n \quad [\ni \varepsilon]$
2. $U^+ = \bigcup_{n \geq 1} U^n$