

Theoretische Informatik

Skript zur Vorlesung

Carsten Damm

Stand: 14. April 2025

Inhaltsverzeichnis

Vorbemerkungen	1
1 Einführung	3
1.1 Bücher und Lehrmaterial	3
1.2 Motivation	5
1.3 Symbole, Wörter und Sprachen	7
1.4 Operationen auf Sprachen	10
1.5 Sprachklassen	14
1.6 Reguläre Ausdrücke	16
2 Endliche Automaten und reguläre Sprachen	17
2.1 Definition endlicher Automaten	17
2.2 Reguläre Sprachen	23
2.3 Automatenminimierung	28
2.4 Nichtdeterministische endliche Automaten	34
2.5 Abschlusseigenschaften	41
2.6 Reguläre Grammatiken	45
2.7 String Matching mit endlichen Automaten	50
2.8 Entscheidungsprobleme für reguläre Sprachen	52
3 Kontextfreie Sprachen und Kellerautomaten	54
3.1 Chomsky-Typen	54
3.2 Eigenschaften kontextfreier Grammatiken	55
3.3 Chomsky-Normalform und Pumping-Lemma	60
3.4 Der CYK-Algorithmus	65
3.5 Abschlusseigenschaften	67
3.6 Kellerautomaten (Pushdown-Automaten)	70
3.7 Entscheidungsprobleme für kontextfreie Grammatiken	76
4 Typ1 und Typ0-Sprachen, Turing-Maschinen	77
4.1 Kontextsensitive Sprachen	77
4.2 Turing-Maschinen	80
4.3 Akzeptor-Charakterisierungen von Typ 1- und Typ 0-Sprachen	85
4.4 Einschub: Einige technische Vorbereitungen	89
4.5 Rekursiv aufzählbare Sprachen	91
4.6 Überblick/Zusammenfassung	94
5 Berechenbarkeit	96
5.1 Intuitive Berechenbarkeit	96
5.2 Turing-Berechenbarkeit	99
5.3 LOOP-, WHILE- und GOTO-Berechenbarkeit	106
5.4 Primitiv und μ -rekursive Funktionen	115
5.5 Die Churchsche These	122
5.6 Die Ackermann-Funktion	123
5.7 Entscheidbarkeit und Semi-Entscheidbarkeit	126

5.8	Reduzierbarkeit und Unentscheidbarkeit	131
-----	--	-----

Standardlehrbücher für eine Einführung in die Theoretische Informatik sind: **[Schöning]** *Theoretische Informatik — kurz gefasst* (Spektrum Akademischer Verlag, 5. Aufl. 2008) und **[Hoffmann]** *Theoretische Informatik* (Carl Hanser Verlag, 4. Auflage 2018). Im wesentlichen folgt die Vorlesung diesen, greift aber zusätzlich Ideen aus **[Gopalakrishnan]** *Automata and Computability: A programmer's perspective*. (CRC Press, 1. Aufl. 2019) auf, theoretische Konzepte anhand von Python zu demonstrieren. Um davon zu profitieren, müssen Sie Python gar nicht beherrschen — es geht, wie Sie sehen werden, nur darum, kurze Codeschnipsel zu verstehen. Dafür genügt etwas Grundbildung in irgendeiner gängigen Programmiersprache¹.

Änderungen gegenüber Version vom 4. Juli (Stand: 8. Juli 24)

5.7-8 und 5.8-4 das Fokuszeichen auf *Band 1* ist gemeint (nicht auf Band 2)

Änderungen gegenüber Version vom 25. Juni (Stand: 4. Juli 24)

5.7-5 Bemerkung über geeignete Codierungen (nur in der Skript-Version sichtbar, nicht auf den Folien) etwas vorsichtiger formuliert

5.7-6 Die vorige Fassung des Beweises beruhte tatsächlich auf dem *allgemeinen* Halteproblem. Hier geht es jedoch um das *spezielle* - dieser Lapsus ist nun behoben.

5.7-9 kleine Tippfehler korrigiert

Änderungen gegenüber Version vom 29. Mai (Stand: 25. Juni 24)

4.5-2 Korrektur des Semi-Entscheidungers (letzte beiden Code-Zeilen)

5.3-21 Stellenwertbasis $b = m + 1$

Änderungen gegenüber Version vom 15. Mai (Stand: 29. Mai 24)

4.2-7 Die Konfigurationsfolge zur Eingabe $x = 101$ ist jetzt korrigiert, und als zweites Beispiel wird die Konfigurationsfolge zur Eingabe $x = 11$ gezeigt.

Änderungen gegenüber Version vom 15. April (Stand: 15. Mai 24)

1.3-7 Definition „lexikografisch kleiner“ korrigiert (es fehlte der Fall, dass u Präfix von v ist)

1.3-8 Aufzählung: Nummerierung beginnt bei 1, darum bildet f von \mathbb{N}_+ nach M ab (nicht von \mathbb{N})

¹Wenn Sie Spaß daran haben, selbst aktiv mit dem Code zu experimentieren, folgen Sie im Stud.IP-Wiki den Hinweisen zu "Jove".

- 2.6-4, 3.2-1, -5, -6** in Grammatik-Beispielen für arithmetische Ausdrücke bzw. für Multiplikationsterme wurde $*$ als Terminalsymbol für das Multiplikationszeichen benutzt, was zu Verwechslungen mit dem Metasymbol $*$ (Kleene-Stern) führen kann. Das ist nun korrigiert: als Multiplikationszeichen wird jetzt \cdot verwendet.
- 3.3-1, 3.3-2** Umwandlung in Chomsky-Normalform korrigiert und jetzt klarer beschrieben
- 3.6-13** Zeile 2 im Beweis (erster Teil der Definition von Δ) korrigiert: es darf *keine Reflexion* verwendet werden (Dank an F. Esser)
- 3.7-1** Notation $\text{Term}(G)$ eingeführt
- 3.7-3** Definition des „Produktionsgraphen“ korrigiert (Dank an F. Esser)

1 Einführung

1.1 Bücher und Lehrmaterial

Lehrbücher

1.1-1

Folien+Skript etc. werden bereitgestellt, aber falls Sie anhand eines Buches lernen wollen, sind die ersten beiden sicher empfehlenswert:

Uwe Schöning: Theoretische Informatik — kurz gefasst



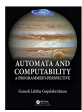
deckt den kompletten Vorlesungsstoff ab, kommt schnell zur Sache, gut lesbar

Dirk W. Hoffmann: Theoretische Informatik



etwas andere Themenauswahl (das Wichtigste ist aber dabei), anschaulich und weniger formal

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*)

Dem Python-Ansatz werden wir hier zum Teil folgen.

Python und Theorie?

1.1-2

Muss ich Python können, um dem Kurs zu folgen?

Nein! Python wird hier benutzt, Algorithmen verständlich aber präzise zu beschreiben. Wer *irgendeine* imperative 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.

Muss ich die Jove-Installation genau so machen, wie im Stud.IP-Wiki angegeben? Muss ich das überhaupt benutzen?

Nein, nur wenn Sie es nützlich oder interessant finden, selbst damit zu experimentieren. In diesem Fall brauchen Sie irgendeine Installation. Im Stud.IP finden Sie einen Vorschlag.

1.1-3 Lehrmaterial

Folien und Skript

- beides über Stud.IP
- im wesentlichen gleicher Inhalt, aber das Skript ist an einigen (wenigen) Stellen etwas ausführlicher

Jupyter-Notebooks

- werden über <https://gitlab.gwdg.de> bereitgestellt und gelegentlich aktualisiert, Erklärung dazu im Stud.IP
- bei eigener Installation (siehe oben) nutzbar für eigene Experimente

Zur Unterstützung: Vorlesungsaufzeichnungen

- sind am Schreibtisch (d.h. ohne Hörer) erstellt
- werden über Stud.IP bereitgestellt

1.1-4 Übungen und Prüfungen

Übung

- Übungsaufgaben und Hinweise zur Übung werden über Stud.IP bekannt gegeben
- Bearbeitung der Übungsaufgaben ist relevant für die Klausurzulassung - genaue Regelung wird noch bekannt gegeben

Prüfung in der vorlesungsfreien Zeit

- genauer Termin und Prüfungsform: siehe Vorlesungsankündigung (in Stud.IP verlinkt)
- Teilnahme erfordert, dass Sie sich bei Flexnow anmelden und zwar
 - sowohl zur Übung (für die Prüfungsvorleistung)
 - als auch zur Klausur

Anmeldefristen: siehe Vorlesungsankündigung

1.2 Motivation

Syntaxbeschreibungen

1.2-1

- *Bezeichner* bestehen aus Buchstabe, optional gefolgt von Ziffern, und Buchstaben. Beispiel: `Head1_ptr`
- *Variablendeklarationen* 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; }
```

Automatische Syntaxprüfung?

Wie soll das aufgrund solcher rein sprachlicher und beispielhafter Vorgaben funktionieren?

Compilation in den Pioniertagen der Programmierung

1.2-2

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

1.2-3

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

Kontextsensitive Muster

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

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

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

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.

1.2-4 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.

1.3 Symbole, Wörter und Sprachen

Symbol und Alphabet


1.3-1

Symbol (Zeichen)

= kleinste bedeutungstragende Einheiten der Kommunikation. Bedingung: Unterscheidbarkeit von anderen Zeichen.

Beispiele: 0, 1, a, ...

Sogar Grafiken oder Wörter/Wortgruppen können Zeichen sein, wenn sie stellvertretend für irgendeinen „Sinn“ stehen.

Beispiele: ⚡, , STOP, Straßenbahn kreuzt, ...

Alphabet

Ein **Alphabet** ist eine nichtleere, endliche Menge von Symbolen.

Wir benutzen Σ als Bezeichnung(!) für ein vereinbartes Alphabet. Kommen weitere ins Spiel, so können wir sie mit $\Sigma_1, \Sigma_2, \Sigma', \dots$ unterscheiden.

Wörter und Operationen auf Wörtern

1.3-2

Wörter

Ein **Wort** (**String**) über Σ ist eine endliche Folge w von Zeichen aus Σ und wird ohne Trennzeichen q notiert. Die Länge von w wird mit $|w|$ bezeichnet. Beispiel: 101001 hat Länge 6. Das **leere Wort** ε ist die Zeichenfolge der Länge 0.

Konkatenation von Wörtern

Die **Konkatenation** (Verkettung) von Wörtern $u, v \in \Sigma$ ist die Operation $(u, v) \mapsto uv$, wobei $uv = u \cdot v$ die Hintereinanderschreibung angibt. Offenbar gilt $|uv| = |u| + |v|$ und $u\varepsilon = \varepsilon u = u$ für alle Wörter u, v .

Potenzierung

Die ***n*-te Potenz** u^n eines Strings u ist die Konkatenation von n Exemplaren von u . Potenzen beziehen sich nur auf den unmittelbar davor stehenden Bestandteil.

Beispiel: $0001^3 = 000111$, $(0001)^3 = 000100010001$

Spezialfall: $u^0 = \varepsilon$ für alle Wörter u .

1.3-3 Python-Notation

Begriff	formelhaft	Python
String	$u = 010010001, v = 00$	<code>u = '010010001', v = '00'</code> oder <code>u = "010010001", v = "00"</code>
Länge	$ u $	<code>len(u)</code>
Zeichen	$0, 1, a, \dots$	<code>'0', '1', 'a', \dots</code> (Strings der Länge 1)
leeres Wort	ε	<code>"</code>
Alphabet	$\{0, 1\}$	<code>set({'0', '1'})</code>
Konkatenation	$uv = u \cdot v = 01001000100$	<code>u + v == '01001000100'</code>
Potenzierung	u^3	<code>u * 3</code>

siehe Jupyter-Notebook
1.3-4

Teilwörter

Sei w ein Wort über Σ .

Ein Wort y heißt **Teilwort** (**Teilstring**, **Faktor** oder **Infix**) von w , falls es Wörter x, z gibt mit $w = xyz$.

Präfix und Suffix

y heißt dann **Präfix** (bzw. **Suffix**) von w , falls darüberhinaus $x = \varepsilon$ (bzw. $z = \varepsilon$).

SoSe2020 hat z.B. So als ein Präfix, e2020 als ein Suffix und oSe als Infix.

Ein Infix/Präfix/Suffix y von w heißt **echt**, falls außerdem $y \neq w$.

1.3-5 Sprachen

Kleenesche Hülle

Die Menge aller Wörter über Σ (inklusive ε) wird mit Σ^* bezeichnet (**Kleenesche Hülle** von Σ). Die Menge der *nichtleeren Wörter* über Σ wird mit Σ^+ bezeichnet (**positive Hülle**). Es gilt: $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$

Formale Sprache

Jede Teilmenge $L \subseteq \Sigma^*$ wird **formale Sprache** über Σ genannt. Die Menge kann *leer*, endlich oder unendlich sein.

Beispiele

Alphabet Σ	Sprache	Beispielwörter	Größe
beliebig	Zero = \emptyset	–	0
beliebig	Unit = $\{\varepsilon\}$	ε	1
$\{a, b, c\}$	MyLang = $\{\varepsilon, aa, aba\}$	ε, aa	3
$\{a, \dots, z\}$	Σ^*	b	∞
$\{a, b, c\}$	$\text{Pref}(abc) = \{w : w \text{ Präfix von } abc\}$	ε, a, ab, abc	3
$\{0, 1\}$	$L_U = \{0^i 1 : i \geq 0\}$	$1, 0001, \underbrace{00\dots 0}_{100} 1$	∞
$\{a, b\}$	$L_{\#a < \#b} = \{a^i b^j : i, j \geq 0 \wedge i < j\}$	$b, abb, aabbb, \dots, a^{50} b^{55}$	∞

Sprachen mit Python definieren

1.3-6

- endliche Sprachen können wir durch Auflisten aller Wörter angeben:

```
MyLang = {'', 'aa', 'aba'}
Zero = set() # leere Menge = leere Sprache
Unit = set({''}) # Sprache, die nur das leere Wort enth lt
```

- unendliche Sprachen können wir auf diese Weise höchstens teilweise angeben, z.B. bis zu einer bestimmten Länge

```
L01Rep9 = {'01'*i for i in range(10)} # range(10) = {0,1,2,3,4,5,6,7,8,9}
La_lt_b_9 = {'a'*i + 'b'*j for i in range(10) for j in range(i)}
```

Lexikographische Ordnung auf Σ^*

1.3-7

Sei $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_\ell\}$ mit einer Ordnung auf den Zeichen: $\sigma_1 < \sigma_2 < \dots < \sigma_\ell$.

Nun definieren wir „Wort u ist **lexikographisch kleiner als** Wort v ($u \prec v$)“:

- $u = u_1 u_2 \dots u_m \prec v = v_1 v_2 \dots v_n$, falls u Präfix von v ist oder aber $\exists i : u_i < v_i \wedge \forall j < i : u_j = v_j$ (d.h. die Zeichen an der ersten Unterscheidungsstelle stehen in Alphabetordnung)

Python

```
def lexlt(s, t):
    if t == '':
        return False
    if s == '' or s[0] < t[0]:
        return True
    return (s[0] == t[0]) & lexlt(s[1:], t[1:])
```

Problem

Lexikographische Anordnung ist ungeeignet für systematische Auflistungen:

$$\{0, 1\}^* = \{\varepsilon, 0, 00, 000, 0000, 00000, \dots\} ???$$

Numerische Ordnung auf Σ^*

1.3-8

längen-lexikographisch kleiner als ($<$)

- falls $|u| < |v|$, so $u < v$

- falls $|u| = |v|$, so $u < v$ gdw. $u \prec v$

Beispiel: $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$

Längen-lexikographisch = numerisch

Eine Abbildung $f : \mathbb{N}_+ \rightarrow M$ mit $M = \{f(1), f(2), \dots\}$ heißt **Aufzählung** von M .

Beispiel: $\{0, 1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$ entspricht der Aufzählung

$$n \mapsto \text{Binärdarstellung von } n \text{ ohne führende 1.}$$

Anders ausgedrückt: für das n -te Wort w gilt $1w = \text{bin}(n)$.

Python

```
# Python-Funktion bin, Bsp.: bin(5) == '0b101'
def nthbitstring(n):
    return bin(n)[3:] # Praefix '0b1' weglassen
```

Bei Aufzählungen dürfen Elemente mehrfach auftreten (z.B. $f(1) = f(2)$). Falls M unendlich viele Elemente enthält und durch f aufgezählt wird, so ist die folgende Abbildung $g : \mathbb{N} \rightarrow M$ sogar eine *bijektive* Aufzählung: $g(n) := f(k)$, wobei k definiert ist durch $|\{f(1), \dots, f(k-1)\}| = n-1 \wedge |\{f(1), \dots, f(k)\}| = n$. Unendliche Mengen, die bijektiv aufzählbar sind, heißen in diskreter Mathematik/Mengenlehre *abzählbar unendlich*.

1.4 Operationen auf Sprachen

1.4-1 Produkt zweier Sprachen

= Konkatenationen der Wörter der ersten mit denen der zweiten Sprache:

$$L_1 \circ L_2 = L_1 L_2 := \{xy : x \in L_1 \wedge y \in L_2\}$$

Python-Funktion

```
def cat(L1, L2):
    return set({x+y for x in L1 for y in L2})
```

1.4-2 Potenzierung von Sprachen

Die leere Sprache \emptyset

Für alle Sprachen L gilt: $L \circ \emptyset = \emptyset \circ L = \emptyset$.

Vergleich: $\forall x \in \mathbb{N} : 0 \cdot x = x \cdot 0 = 0$

Die Einheitssprache $\{\varepsilon\}$

Für alle Sprachen L gilt: $L \circ \{\varepsilon\} = \{\varepsilon\} \circ L = L$.

Vergleich: $\forall x \in \mathbb{N} : 1 \cdot x = x \cdot 1 = x$

Das motiviert diese Festlegungen

- $L^0 := \{\varepsilon\}$
- $L^n := LL^{n-1}$ für $n > 0$

Insbesondere

$$\emptyset^i = \begin{cases} \{\varepsilon\} & i = 0 \\ \emptyset & \text{sonst.} \end{cases}$$

```
def power(L,n):  
    return Unit if n==0 else cat(L,power(L,n-1))
```

Hülle einer formalen Sprache

1.4-3

Ist $L \subseteq \Sigma^*$, so ist die **Kleenesche Hülle** (auch Kleene-Stern von L definiert durch

$$L^* = \bigcup_{i \geq 0} L^i.$$

Die **positive Hülle** von L ist die Menge

$$L^+ = \bigcup_{i > 0} L^i.$$

Beispiel

Sei $L = \{0, 11\}$. Dann enthält L^* diese Wörter

$\varepsilon, 0, 00, 11, 000, 011, 110, 0000, 0011, 0110, 1100, 1111, \dots$

und $L^+ = L^* \setminus \{\varepsilon\}$.

Einfache Mengenoperationen

1.4-4

Seien L, L_1, L_2 formale Sprachen über Σ .

Vereinigung und Schnitt

$$L_1 \cup L_2 = \{w : w \in L_1 \vee w \in L_2\}$$

$$L_1 \cap L_2 = \{w : w \in L_1 \wedge w \in L_2\}$$

```
def lunion(L1,L2):  
    return L1 | L2  
def lint(L1,L2):  
    return L1 & L2
```

Differenz, symmetrische Differenz und Komplement

$$\begin{aligned} L_1 \setminus L_2 &= \{w : w \in L_1 \wedge w \notin L_2\} \\ L_1 \Delta L_2 &= (L_1 \setminus L_2) \cup (L_2 \setminus L_1) \\ \bar{L} &= \Sigma^* \setminus L \end{aligned}$$

```
def lminus(L1,L2):          # Differenz
    return L1 - L2
def lsymdiff(L1,L2):       # symmetrische Differenz
    return L1 ^ L2
def lcomplem(L,alphabet,n): # n-te Approximation des Komplements
    return lminus(star(alphabet), L)
```

1.4-5 Beispiele

- Felder des Schachbretts

$$\{A, B, C, D, E, F, G, H\} \circ \{1, 2, 3, 4, 5, 6, 7, 8\}$$

- Uhrzeiten

$$(\{2\} \circ \{0, 1, 2, 3\} \cup \{\varepsilon, 1\} \circ \Sigma) \circ \{:\} \circ \{0, 1, \dots, 5\} \circ \Sigma, \text{ mit } \Sigma = \{0, 1, \dots, 9\}$$

- Schaltjahre² (Jahr durch 400 teilbar oder durch 4 aber nicht durch 100)

$$S = VH \cup (V \cap \overline{\Sigma^* \circ \{00\}}),$$

wobei $V = G_1 \cup \Sigma^2 \circ (G \circ G_1 \cup U \circ G_2)$, $VH = V \circ \{00\}$, mit
 $G_1 = \{0, 4, 8\}$, $G_2 = \{2, 6\}$, $G = G_1 \cup G_2$, $U = \bar{G}$

1.4-6 Σ^* als Monoid

Ein **Monoid** (M, \circ, e) besteht aus einer Menge M mit assoziativer Verknüpfungsoperation $\circ : M \times M \rightarrow M$ und neutralem Element $e \in M$, d.h.

- $\forall a, b, c \in M : a \circ (b \circ c) = (a \circ b) \circ c =: a \circ b \circ c$
- $\forall a \in M : a \circ e = e \circ a = a$.

Beispiele

- $(\Sigma^*, \cdot, \varepsilon)$
- $(\mathbb{N}, +, 0)$ — natürliche Zahlen mit Addition
- $(\mathbb{N}_+, \cdot, 1)$ — positive natürliche Zahlen mit Multiplikation

Freies Monoid

Ein Monoid heißt **frei erzeugt** von $A \subseteq M$, wenn jedes Element *eindeutig* darstellbar ist als Verknüpfung $a_1 \circ a_2 \circ \dots \circ a_n$ endlich vieler Elemente aus A .

²vierstellig

- $(\Sigma^*, \cdot, \varepsilon)$ ist frei erzeugt von Σ
- $(\mathbb{N}, +, 0)$ ist frei erzeugt von $\{1\}$
- $(\mathbb{N}_+, \cdot, 1)$ ist *nicht* frei erzeugt, z.B. $2 \cdot 3 = 3 \cdot 2$

Homomorphismen

1.4-7

Eine Abbildung $h : \Sigma^* \rightarrow \Gamma^*$ heißt **Homomorphismus**, falls

$$\forall u, v \in \Sigma^* : h(uv) = h(u)h(v).$$

Willkürlich gewählte Abbildungen haben diese Eigenschaft nicht. Beispiel:
 $\Sigma = \Gamma = \{0, 1\}, h(0) := 0, h(1) = 1$ falls $w \neq 0$.

Universalität

Sei $h : \Sigma \rightarrow \Gamma^*$ beliebig. Da Σ^* frei ist, gibt es genau eine **Fortsetzung** von h zu einem Homomorphismus $\hat{h} : \Sigma^* \rightarrow \Gamma^*$:

$$\hat{h}(w) = \begin{cases} \varepsilon & \text{falls } w = \varepsilon \\ h(w_1)h(w_2) \cdots h(w_n) & \text{falls } w = w_1w_2 \cdots w_n \wedge \forall i w_i \in \Sigma. \end{cases}$$

Konvention

Die Fortsetzung \hat{h} wird meist ebenfalls mit h bezeichnet.

Beispiele

1.4-8

Sei $\Sigma = \{0, 1, 2\}$ und $\Gamma = \{a, b\}$. Wir definieren Homomorphismen h, g durch Fortsetzung:

- Sei $h : \Sigma^* \rightarrow \Gamma^*$ festgelegt durch

$$0 \mapsto a, 1 \mapsto b, 2 \mapsto b.$$

Dann ist z.B. $h(0012) = aabb$ und das **Bild** von $L_1 = \{0^n 1 2^{n-1} : n \geq 1\}$ unter h ist

$$h(L_1) = \{a^n b^n : n \geq 1\}.$$

- Sei $g : \Sigma^* \rightarrow \Sigma^*$ festgelegt durch

$$0 \mapsto 0, 1 \mapsto 10, 2 \mapsto 0.$$

Dann ist $g(021) = 0010$ und
 $g^{-1}(0010) = \{021, 001, 221, 201\} = \{0, 2\}^2 \{1\}$ und
 $g^{-1}(0^n 10^n) = \{0, 2\}^n \{1\} \{0, 2\}^{n-1}$.

Für das **Urbild** $g^{-1}(L_2)$ von $L_2 = \{0^n 10^n : n \geq 1\}$ unter g gilt

$$g^{-1}(L_2) \cap \{0\}^* \{1\} \{2\}^* = \{0^n 1 2^{n-1} : n \geq 1\}.$$

Reflexion

1.4-9

Reflexion eines Worts

Für $w \in \Sigma^*$ ist w^R rekursiv definiert durch

$$w^R = \begin{cases} \varepsilon & \text{für } w = \varepsilon \\ v^R a & \text{falls } w = av \text{ für } a \in \Sigma, v \in \Sigma^* \end{cases}$$

Die Reflexion einer Sprache

$$L^R := \{w^R : w \in L\}$$

```
def revs(s):          # String s umkehren
    return s[::-1]
def revl(L):          # jeden String aus L umkehren
    return set({revs(x) for x in L})
```

Frage

Ist Reflexion $w \mapsto w^R$ ein Homomorphismus?

1.5 Sprachklassen

1.5-1 Stufenaufbau der Mengenlehre

Das Barbier-Paradoxon (Bertrand Russel, 1872–1970)

Den Barbier rasiert genau die Leute, die sich nicht selbst rasieren.
Rasiert der Barbier sich selbst?

Konsequenz: Es entstehen grundsätzliche Probleme, wenn man *uneingeschränkte* Mengenbildungen zulässt:

- Sei R die „Menge“ aller Mengen, die sich nicht selbst als Element enthalten: $R = \{x \mid x \notin x\}$. Enthält R sich selbst, d.h. gilt $R \in R$?

Ausweg: Mengenbildung nur aus Elementen „niederer Stufe“

Wir betrachten ein Universum von „Urelementen“. Mengen *erster Stufe* sind Mengen, deren Elemente Urelemente sind. Mengen *zweiter Stufe* (im folgenden **Klassen** genannt) haben als Elemente Mengen erster Stufe, usw.

Beispiel (fixiertes Alphabet Σ und Universum Σ^)*

- Mengen erster Stufe = formale Sprachen über Σ
- Klassen = bestimmte Mengen von formalen Sprachen über Σ , z.B.
 - Klasse Fin_Σ der endlichen Sprachen über Σ
 - Klasse $\varepsilon\text{-Free}_\Sigma$ der Sprachen über Σ , die ε nicht enthalten.

Formale Sprachklassen

1.5-2

Die Zeichen praktisch aller Schriftsprachen sind im Unicode-Zeichensatz enthalten. Unicode wird ständig weiterentwickelt und ist im Prinzip beliebig erweiterbar.

Ähnlich gehen wir von nun an davon aus, dass alle Alphabete, die wir betrachten, Teilmengen einer abzählbar unendlichen *universellen Symbolmenge* $\Gamma_S = \{z_1, z_2, z_3, \dots\}$ sind³.

Definition

Eine **formale Sprachklasse** ist eine *nichtleere* Klasse \mathcal{L} von formalen Sprachen, so dass folgende Eigenschaften erfüllt sind:

- für jedes $L \in \mathcal{L}$ gibt es ein endliches Alphabet $\Sigma \subseteq \Gamma_S$ mit $L \subseteq \Sigma^*$,
- es gibt ein $L \in \mathcal{L}$ mit $L \neq \emptyset$.

Beispiele für formale Sprachklassen

- Klasse *Fin* der endlichen Sprachen
- Klasse *ε -Free* der ε -freien Sprachen

Reguläre Mengen

1.5-3

Die betrachteten *Sprachoperationen* (einfache Mengenoperationen, Produkt, Hülle und Bild/Urbild unter Homomorphismen und Reflexion) bilden (Paare von) Sprachen auf Sprachen ab.

Eine Sprachklasse \mathcal{C} heißt **abgeschlossen** unter einer Sprachoperation, wenn ausgehend von Sprachen aus \mathcal{C} , das Ergebnis ebenfalls in \mathcal{C} liegt.

Definition

Die Klasse *Reg* der **regulären Mengen** ist die kleinste Sprachklasse, die *Fin* enthält und abgeschlossen ist unter endlich vielen Anwendungen von Vereinigung, Konkatenation und Kleenescher Hülle.

Das Wortproblem

1.5-4

Das **Wortproblem** besteht darin, zu entscheiden, ob ein gegebenes Wort w zu einer gegebenen formalen Sprache L gehört.

Reguläre Mengen haben eine endliche Beschreibung

Um das Wortproblem algorithmisch zu untersuchen, muss L durch eine endliche Beschreibung gegeben sein. Nach Definition ist dies zum Beispiel für reguläre Mengen möglich.

Reg ist sehr robust

1.5-5

Wir werden zeigen: Die Klasse ist abgeschlossen unter *allen* bisher genannten Sprachoperationen. Darüber hinaus auch unter diesen:

³Für uns ist nur wichtig, dass es eine universelle Symbolmenge gibt, nicht, welche es genau ist.

Quotient bzgl. Wort

Sei $L \subseteq \Sigma^*$ eine formale Sprache und $x \in \Sigma^*$ ein Wort. **Rechts-** bzw. **Linksquotient** von L bzgl. x sind definiert durch

$$L/x := \{z \in \Sigma^* : zx \in L\} \text{ bzw. } x \backslash L := \{z \in \Sigma^* : xz \in L\}.$$

Quotient bzgl. Sprache

Sei $K \subseteq \Sigma^*$ eine weitere formale Sprache. Der **Quotient** der beiden Sprachen ist definiert durch

$$L/K := \{x \in \Sigma^* : \exists z \in K \wedge xz \in L\}.$$

Shuffle-Produkt

Das **Shuffle-Produkt** von $L, K \subseteq \Sigma^*$ ist definiert durch

$$L \# K := \{x_1 z_1 x_2 z_2 \dots x_n z_n : \exists n \in \mathbb{N}, x_1, \dots, x_n, z_1, \dots, z_n \in \Sigma^*, x_1 x_2 \dots x_n \in L, z_1 z_2 \dots z_n \in K\}.$$

1.6 Reguläre Ausdrücke

1.6-1 Wie notiert man eine reguläre Menge?

Wir wissen nun über reguläre Mengen:

Definition Sie werden aus endlichen Mengen gebildet durch einfachste Sprachoperationen (Vereinigung, Produkt und Stern)

Robustheit Die Klasse der regulären Mengen ist abgeschlossen unter vielen weiteren Sprachoperationen.

Die Robustheit macht *Reg* zu einer sehr vielseitigen Sprachklasse.

Für bequeme Anwendung brauchen wir aber noch eine kompaktere Notation anstelle vieler Mengenklammern und Operationszeichen, Beispiel:

$$(\{2\} \circ \{0, 1, 2, 3\} \cup \{\varepsilon, 1\} \circ \{0, 1, \dots, 9\}) \circ \{:\} \circ \{0, 1, \dots, 5\} \circ \{0, 1, \dots, 9\}$$

1.6-2 Reguläre Ausdrücke

Syntax: reguläre Ausdrücke (r.A. oder auch r.e. — regular expressions)

Sei Σ ein beliebiges Alphabet, das die **Metasymbole** $\emptyset, \varepsilon, (,), |, *$ nicht enthält.

- \emptyset und ε sind reguläre Ausdrücke.
- Für jedes $\sigma \in \Sigma$ ist σ ein regulärer Ausdruck.
- Sind r und s reguläre Ausdrücke, so auch $(r|s)$, (rs) und (r^*) .

Semantik: *durch regulären Ausdruck beschriebene reguläre Menge*

\emptyset , ε und σ für $\sigma \in \Sigma$ beschreiben die Mengen $\emptyset, \{\varepsilon\}, \{\sigma\} \subset \Sigma^*$.

Beschreiben die regulären Ausdrücke r, s die regulären Mengen M_r, M_s , so beschreiben $(r|s)$, (rs) und (r^*) die Mengen $M_r \cup M_s$, $M_r \circ M_s$ und M_r^* . Durch Assoziativität und Vorrang ($*$ vor \cdot , \cdot vor $|$) können Klammern entfallen.

In Vorgriff auf einen Satz aus dem nächsten Kapitel wird $L(r)$ wird auch kurz „die Sprache von r “ genannt

Beispiele

- Uhrzeiten: $(2(0|1|2|3)|(\varepsilon|1)(0|1|\dots|9)) : (0|1|\dots|5)(0|1|\dots|9)$
- Bitstrings mit 1 an vor-vorletzter Stelle: $(1 + 0) * 1(1 + 0)(1 + 0)$

Erklärung: oft wird $+$ anstelle $|$ verwendet (auch hier im Skript, aus dem Kontext wird klar, was gemeint ist).

Reguläre Ausdrücke und Homomorphismen

1.6-3

Satz (Abschluss unter Homomorphismen)

\mathcal{Reg} ist abgeschlossen unter Homomorphismen, d.h. das Bild $h(M)$ einer regulären Menge $M \subseteq \Sigma^*$ unter einem Homomorphismus $h : \Sigma \rightarrow \Gamma^*$ ist auch eine reguläre Menge.

Beweisskizze (mittels regulärer Ausdrücke)

Sei r regulärer Ausdruck über Σ und $h(r)$ der daraus entstehende r.A., indem jedes Zeichen $\sigma \in \Sigma$ durch $h(\sigma)$ ersetzt wird. Es genügt, $h(M_r) = M_{h(r)}$ für alle r zu beweisen. Das gelingt durch **strukturelle Induktion**.

Basisfälle:

1. $h(M_\emptyset) = h(\emptyset) = \emptyset = M_\emptyset = M_{h(\emptyset)}$
2. $h(M_\varepsilon) = h(\{\varepsilon\}) = \{\varepsilon\} = M_\varepsilon = M_{h(\varepsilon)}$
3. Sei $\sigma \in \Sigma$ und $h(\sigma) = \gamma_1 \cdots \gamma_n$ mit $\gamma_i \in \Gamma$. Dann gilt
$$h(M_\sigma) = h(\{\sigma\}) = \{h(\sigma)\} = \{\gamma_1 \cdots \gamma_n\} = M_{\gamma_1} \cdots M_{\gamma_n} = M_{\gamma_1 \cdots \gamma_n} = M_{h(\sigma)}$$

Induktionsschritt: (wenn Aussage für r, s gilt, dann auch für daraus zusammengesetzte Ausdrücke) — hier nur für $r|s$ gezeigt, Rest analog.

$$h(M_{r|s}) = h(M_r \cup M_s) = h(M_r) \cup h(M_s) = M_{h(r)} \cup M_{h(s)} = M_{h(r)|h(s)} = M_{h(r|s)}$$

□

2 Endliche Automaten und reguläre Sprachen

2.1 Definition endlicher Automaten

Motivation

2.1-1

Grundidee: „Finite state machine“

- endliche viele Zustände, darunter ein Startzustand
- Einwirkungen von außen bewirken Zustandsänderung

Anwendungen

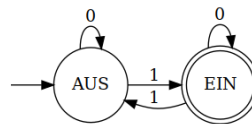
- Getränkeautomat (Münze, Getränk/Abbruch wählen, entnehmen)
- Handlungsabläufe: Bedienungsanleitung, Notfallplan
- Ampelschaltung: Farbwechsel nach Zeitablauf (ggf. angeregt durch Fußgängersignal, beeinflusst durch Verkehrsdichte)
- Textanalyse: ist *dieses Wort* enthalten?, ...
- Bestandteil von Compilern: Lexer — extrahiert und klassifiziert Tokens aus dem Quelltext (Bezeichner, Schlüsselwörter, Literale, ...)

usw.

Endliche Automaten in theoretischer Informatik

erkennende Automaten: keine Ausgabe, Resultat wird durch erreichten Zustand signalisiert

2.1-2 Informales Beispiel: Ein Eingabe-gesteuerter Kippschalter



Lesen von „1“ löst Schaltvorgang aus, Lesen von „0“ wird ignoriert.

Bestandteile

- *Zustände* (eingekreist) darunter der *Startzustand* (Pfeil von außen)
- *Zustandsübergänge* (Pfeile, markiert mit verarbeitbaren Zeichen)
- *Endzustand* (Doppelkreis)

Das Bild (**Zustandsdiagramm**) zeigt nicht den Schalter selbst, sondern ein *formales Berechnungsmodell* davon.

Arbeitsweise

- beginne im Startzustand
- nacheinander Zeichen verarbeiten und Zustand ändern wie angegeben („bedingte Sprünge“)

- Eingaben, die zu Endzustand führen, heißen **akzeptiert** (ein suggestiverer Name für Endzustand ist **Akzeptierungszustand**)

Die erkannte Sprache

2.1-3

Die vom Automaten **erkannte Sprache** (auch „die Sprache des Automaten“) ist die Menge akzeptierter Eingaben.

Frage

Welche Sprache wird von obigem Automaten erkannt?

Antwort

Menge der Bitmuster mit ungerader Anzahl an Einsen

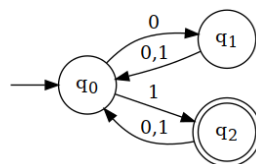
Beobachtung

Für die erkannte Sprache sind die Zustandsnamen unwichtig, anstelle AUS und EIN könnte man genauso gut auch q_0 und q_1 verwenden.

Wichtig sind nur ihre Rollen (Start-, End- oder anderer Zustand) und die Übergänge zwischen ihnen.

Informales Beispiel 2

2.1-4



Frage

Welche Sprache wird von dem Automaten erkannt?

Antwort

- Akzeptierte Wörter müssen eine Länge $2k + 1$ haben und auf 1 enden.
- Jedes solche Wort wird akzeptiert: Nach den ersten $2k$ Zeichen befindet sich der Automat in q_0 . Nur eine 1 am Ende führt zur Akzeptierung.

Vorbereitung: Etwas Mengenlehre

2.1-5

Theoretische Berechnungsmodelle beruhen nur auf mengentheoretischen Begriffen, insbesondere Funktionen und (geordneten) Paaren bzw. n -Tupeln. Letztere benutzen wir als Grundbegriffe, die wir hier nur anschaulich beschreiben. Implizit haben wir die Begriffe (als Folgen von Symbolen) bereits bei der Definition von Wörtern über einem Alphabet benutzt.:

Das **geordnete Paar** (a, b) ist die Zusammenfassung von a und b zu einer Einheit. Allgemeiner ist ein **geordnetes n -Tupel** (a_1, a_2, \dots, a_n) die Zusammenfassung von n Elementen a_1, a_2, \dots, a_n zu einer Einheit. a_1 ist die **erste**, a_2 die zweite, \dots , a_n die n -te **Komponente** des Tupels.

Definition

Das **Kreuzprodukt** (*kartesisches Produkt*) der Mengen A_1, A_2, \dots, A_n ist die Menge

$$A_1 \times \dots \times A_n := \{(a_1, \dots, a_n) : a_1 \in A_1 \wedge \dots \wedge a_n \in A_n\}$$

Umsetzung in Python

```
pair = (a,b) = (4,(5,6,7)) # ein Paar als Einheit aus Zahl und 3-Tupel
pair[0] == a and pair[1] == b == (5,6,7) # Komponentenzugriff
A = {1,2}; B = {3,4} # zwei endliche Mengen und ihr ..
{(x,y) for x in A for y in B} == {(1,3),(1,4),(2,3),(2,4)} #.. Kreuzprodukt
```

2.1-6 Der Begriff der Funktion

Definition

Eine **Funktion** oder *Abbildung* $f : X \rightarrow Y$ aus einer Menge X in eine Menge Y (**Urbild-** und **Bildbereich**) ist eine Teilmenge $f \subseteq X \times Y$ von Paaren (x, y) mit:

das bedeutet, f ist *rechtseindeutig*

Für jedes $x \in X$ gibt es höchstens ein y , so dass $(x, y) \in f$.

Sprechweise: f ordnet x den Funktionswert y zu (Notation: $f(x) = y$ oder $f : x \mapsto y$)

Beispiel zur Realisierung als Python-Dictionary

Voraussetzung: Es gibt nur endlich viele Zuordnungspaare.

```
f = { # Die Zuordnungspaare (x,y) werden in der Form x:y aufgefuehrt:
0:'zero', 1: 'one', # Festlegung der Zuordnung fuer ..
2:'digit', 3:'digit' # .. Elemente 0, 1, 2, 3
}

f[4] = 'digit' # weitere Zuordnung festlegen
b = f[3] # ACHTUNG: auch beim Abruf eckige Klammern [] verwenden
```

Der Urbildbereich von f ist $\{0, 1, 2, 3, 4\}$. Das sind die *Schlüssel* des Dictionaries und *'zero', 'one', 'digits'* sind seine *Werte*.

2.1-7 Formale Beschreibung endlicher Automaten

Definition

Ein **deterministischer endlicher Automat** (*deterministic finite automaton* — DFA) ist ein 5-Tupel $(Q, \Sigma, \delta, q_0, F)$ bestehend aus

- einer endlichen Menge Q (**Zustandsmenge**)

- einem Alphabet Σ (Eingabealphabet)
- einer Funktion $\delta : Q \times \Sigma \rightarrow Q$ (Übergangsfunktion)
- einem Element $q_0 \in Q$ (Startzustand) und
- einer Menge $F \subseteq Q$ (Endzustände, Akzeptierungszustände, ...).

Maschinenlesbare Spezifikation zu Beispiel 2

```
{'Q': {'q1', 'q2', 'q0'},           # 'Q': Wert = eine Menge von Strings
'Sigma': {'0', '1'},              # 'Sigma': Wert = eine Menge von Strings
'Delta': {'q1', '0': 'q0',        # 'Delta': Wert = ein Dictionary
          ('q1', '1'): 'q0',      # Schlüssel:
          ('q2', '0'): 'q0',      # (Zustand, Symbol)-Paare
          ('q2', '1'): 'q0',
          ('q0', '0'): 'q1',
          ('q0', '1'): 'q2'},
'q0': 'q0',                       # 'q0': Wert = ein String
'F': {'q2'}}                      # 'F': Wert = eine Menge von Strings
}
```

siehe Jupyter-Notebook

`D['Delta']` ist selbst ein Dictionary. Zum Beispiel wird der Zustand, in den D nach Lesen von `a` (eine Variable, deren Wert ein Symbol ist) vom Startzustand aus wechselt, so angegeben:

`D['Delta'][(D['q0'], a)]`

Beispiel: Einen String-Klassifizierer konstruieren

siehe Jupyter-Notebook
2.1-8

Aufgabe

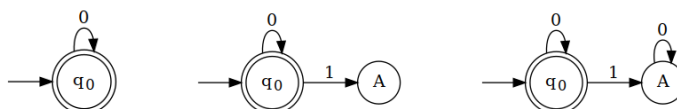
DFA angeben, der die 0/1-Strings mit durch 3 teilbarer Anzahl von 1en akzeptiert, andere *verwirft*.

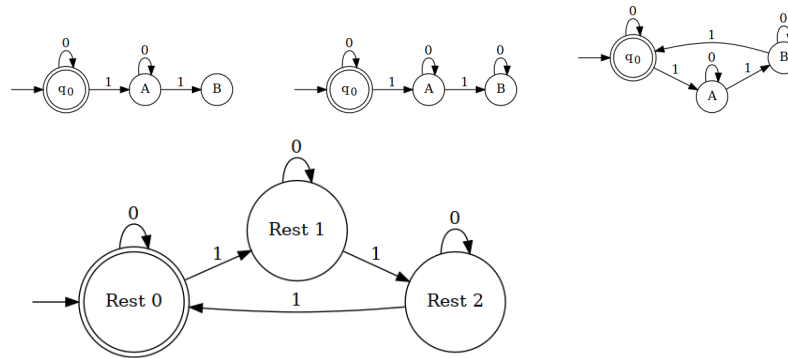
Idee

- bestimme charakteristische akzeptierte Wörter
 - ε , 111 sind die kürzesten, dann 0111, 1011, ...
 - beliebige Wörter der Form $(111)^n$
 - eingestreute 0en „stören nicht“
- da ε akzeptiert wird, ist der Startzustand gleichzeitig auch Endzustand
- ergänze Übergänge, so dass jeder 0/1-String verarbeitet werden kann
- finde klarere Darstellung/Bedeutung der Zustände

Umsetzung der Idee

2.1-9





2.1-10 Partielle und totale Funktionen

Sei $f : X \rightarrow Y$ eine Abbildung. Der **Definitionsbereich** von f ist die Menge der x , für die eine Zuordnung festgelegt ist:

$$D_f = \{x : x \in X \wedge \exists y \in Y \wedge f(x) = y\}.$$

Definition

f heißt **total** (Funktion von X nach Y , Bezeichnung $f : X \rightarrow Y$), falls $D_f = X$.

Anderenfalls heißt f **partiell** (oder Funktion aus X nach Y , Notation $f : X \rightrightarrows Y$).

Beispiele (werden später noch benötigt)

Die Vorgängerfunktion ist eine partielle Funktion $\text{pred} : \mathbb{N} \rightrightarrows \mathbb{N}$. Durch die Festlegung $\text{pred}(0) := 0$ wird sie „totalisiert“.

Analog: Subtraktion wird totalisiert durch $\text{sub}(x, y) := 0$, falls $x \leq y$.

```
def pred(x):
    return x-1 if x>0 else 0      # modifizierter Vorgaenger
def sub(x,y):
    return x-y if x>y else 0      # modifizierte Subtraktion
```

Bemerkung Bei Python-Dictionaries ist der „Definitionsbereich“ genau die Schlüsselmenge. Definitionslücken gibt es also nicht. Aber man kann sie in robuster Weise simulieren — sollte das einmal notwendig sein.

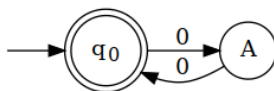
```
f = {0: 'zero', 1: 'one', 2: 'digit', 3: 'digit',
     4: None}
print(f[4]) # erzeugt keine Ausgabe, auch keinen Fehler
print(f[5]) # erzeugt Fehler
x = ... # irgendein Schluessel
if f[x] == None:
    print('Definitionslücke')
```

2.1-11 Unvollständige Automaten und ihre Vervollständigung

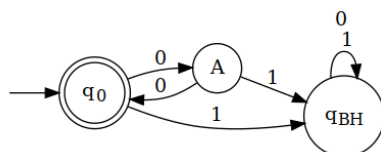
$\delta : Q \times \Sigma \rightarrow Q$ erfordert ein Zustandsdiagramm mit $|Q \times \Sigma|$ Kanten. Partielle Übergangsfunktionen sind oft übersichtlicher: Ist $\delta(q, \sigma)$ nicht definiert, so

wird σ von q aus nicht verarbeitet, und die Eingabe nicht akzeptiert. Zur Unterscheidung nennen wir solche DFAs **partiell**, die anderen **total** oder **vollständig**.

Beispiel



Erkannte Sprache: „gerade Anzahl 0en“.



Für Eingabealphabet $\{0, 1\}$ totalisiert, erkannte Sprache bleibt gleich.

Sackgasse = ein Zustand von dem aus kein Akzeptierungszustand erreichbar ist. Beispiel: q_{BH} ("black hole"). Wird den Definitionslücken eine Sackgasse zugeordnet, ändert sich offenbar die erkannte Sprache nicht:

Lemma (DFA-Vervollständigung)

Durch (eventuelle Hinzunahme von) Sackgassen, kann man jeden partiellen DFA totalisieren, ohne seine Sprache zu ändern.

2.2 Reguläre Sprachen

Formalisierung der Arbeitsweise eines DFA

2.2-1

Fortsetzung der Übergangsfunktion

Sei D ein DFA mit Übergangsfunktion $\delta_D : Q \times \Sigma \rightarrow Q$ und Startzustand q_0 . Die *fortgesetzte* oder **erweiterte Übergangsfunktion** $\hat{\delta}_D : Q \times \Sigma^* \rightarrow Q$ ist definiert durch:

$$\hat{\delta}_D(q, w) = \begin{cases} q & \text{falls } w = \varepsilon \\ \hat{\delta}_D(\delta_D(q, a), v) & \text{falls } w = av, a \in \Sigma. \end{cases}$$

- $q \mapsto \delta_D(q, a)$ beschreibt Verarbeitung von Symbol a beginnend in q
- $q \mapsto \hat{\delta}_D(q, w)$ die Verarbeitung von Wort w beginnend in q

Sei $w = w_1 w_2 \cdots w_n \in \Sigma^*, n \geq 0$. Wir bezeichnen die Zustandsfolge $r(w) := (\hat{\delta}_D(q_0, w_1 \cdots w_i))_{i=1}^n$ als **Lauf** von w (in D).

Ein Zustand q heißt **erreichbar** in D , wenn er in einem Lauf liegt, das heißt, wenn es eine Eingabe w gibt, mit $\hat{\delta}_D(q_0, w) = q$.

Akzeptieren und Erkennen

Sei $D = (Q, \Sigma, \delta, q_0, F)$ ein DFA.

- Wort $w \in \Sigma^*$ wird von D **akzeptiert**, falls $\hat{\delta}(q_0, w) \in F$.
- Die von D **erkannte Sprache** ist definiert durch

$$L_D := \{w : \hat{\delta}(q_0, w) \in F\}.$$

2.2-2 Realisierung in Python

Hier noch einmal das frühere Beispiel:

```
D = \
{
    'Q': {'A', 'F', 'I'},           # 'Q': Wert = eine Menge von Strings
    'Sigma': {'0', '1'},           # 'Sigma': Wert = eine Menge von Strings
    'Delta': {('A', '0'): 'I',     # 'Delta': Wert = ein Dictionary
               ('A', '1'): 'I',
               ('F', '0'): 'I',
               ('F', '1'): 'I',
               ('I', '0'): 'A',
               ('I', '1'): 'F'},
    'q0': 'I',                     # 'q0': Wert = ein String
    'F': {'F'}                    # 'F': Wert = eine Menge von Strings
}
```

Fortsetzung der Übergangsfunktion und Akzeptierung

```
def step_dfa(D, q, a):
    return D['Delta'][q, a]

def run_dfa(D, w, q):
    return q if w == '' else run_dfa(D, w[1:], step_dfa(D, q, w[0]))

def accepts_dfa(D, w):
    return run_dfa(D, w, D['q0']) in D["F"]
```

2.2-3 Reguläre Sprachen

Definition

Sprache $L \in \Sigma^*$ heißt **regulär**, wenn es einen DFA $D = (\Sigma, Q, q_0, \delta, F)$ gibt mit

$$L = L_D.$$

Die Klasse der regulären Sprachen über einem Alphabet Σ wird mit $\text{REG}(\Sigma)$ bezeichnet, die *aller* regulären Sprachen mit REG .

REG ist eine nichttriviale Sprachklasse

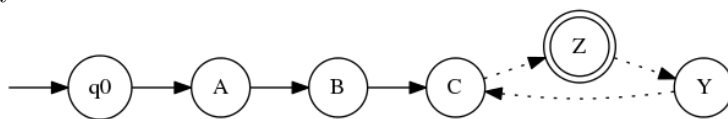
1. Es ist eine nichtleere Sprachklasse (Übungsaufgabe: Beliebige endliche Sprachen sind enthalten. Die Klasse enthält aber auch unendliche Sprachen.)
2. Es gibt auch formale Sprachen, die *nicht* in REG enthalten sind.

Die Zyklenstruktur endlicher Automaten

2.2-4

Beispiel

DFA's über *unärem* Alphabet (d.h. $|\Sigma| = 1$, o.B.d.A. $\Sigma = \{a\}$) enthalten genau einen Zyklus aus erreichbaren Zuständen:



Beobachtung 1

- (1) Ein akzeptiertes „langes Wort“ (Länge $\geq |Q|$) muss auf seinem *Lauf* (siehe Paragraph 2.2-1) mindestens einen Zustand wiederholt besuchen (*Wiederholungszustand*)
- (2) Jede unäre reguläre Sprache ist Vereinigung von Sprachen, die endlich sind oder aber die Form $\{a^k(a^s)^\ell : \ell \geq 0\}$ haben (für geeignete $k \geq 0, s \geq 1$).

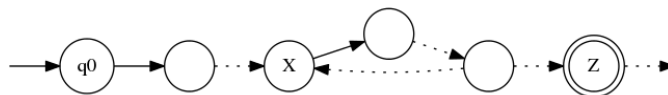
Endliche Automaten über größerem Alphabet

Jeder DFA $(\Sigma, Q, q_0, \delta, F)$ hat Zyklen (betrachte z.B. Einschränkung auf ein unäres Alphabet), insbesondere gilt (1) auch im Fall $|\Sigma| > 1$.

Konsequenzen

2.2-5

Sei w ein akzeptiertes Wort der Länge $n \geq |Q|$, X der erste Wiederholungszustand seines Laufs (das kann auch der Startzustand sein) und $Z = \hat{\delta}(q_0, w)$ sein Akzeptierungszustand (kann auch auf dem Zyklus $X \rightsquigarrow X$ liegen).



Beobachtung 2

w ist zerlegbar in 3 Abschnitte $w = xyz$, wobei

- x = Wort mit Lauf $q_0 \rightsquigarrow X$ (vom initialen Zustand bis zum ersten Wiederholungszustand)
- y = Wort mit Lauf $X \rightsquigarrow X$ (vom ersten Besuch des Wiederholungszustands bis zum zweiten), insbesondere gilt $|y| \geq 1$ und $|xy| \leq n$

- z = Wort mit Lauf $X \rightsquigarrow Z$ (vom zweiten Besuch des Wiederholungszustands bis zum Akzeptierungszustand)

Schlussfolgerung

Die Wörter xz, xy^2z, xy^3z, \dots werden ebenfalls akzeptiert.

2.2-6 Das Pumping-Lemma für reguläre Sprachen

Satz (*Pumping-Lemma für reguläre Sprachen*)

Ist $L \in \text{REG}(\Sigma)$, dann gibt es eine Zahl $n \geq 1$, so dass alle $w \in L$ mit $|w| \geq n$ einen „ n -Pumpfaktor“ $y \in \Sigma^+$ in folgendem Sinne haben:

Es gibt eine Zerlegung $w = xyz$ mit $x, z \in \Sigma^*, |xy| \leq n$ und $xy^\ell z \in L$ falls $\ell \geq 0$.

Kurzform

$$L \in \text{REG}(\Sigma) \Rightarrow \underbrace{\exists n \forall w \in L : |w| \geq n \Rightarrow w \text{ hat einen "n-Pumpfaktor"}}_{\text{Cond}_{\text{REG}}(L)}$$

Beweis

- wegen $L \in \text{REG}$ gibt es einen DFA D mit $L = L_D$
- sei n die Anzahl seiner Zustände
- zerlege w wie beschrieben in die drei Abschnitte gemäß Beobachtung 2 und schlussfolgere die Behauptung □

2.2-7 Anwendung

Mit dem Pumping-Lemma kann man zeigen, dass eine Sprache $L \subseteq \Sigma^*$ *nicht* regulär ist. Dazu benutzt man seine *Kontraposition*:

$$\neg \text{Cond}_{\text{REG}}(L) \Rightarrow L \notin \text{REG}(\Sigma).$$

Die negierte Pumpbedingung in Worten

Die Sprache enthält beliebig lange Wörter ohne Pump-Faktoren.

Genauer: Die negierte Pumpbedingung $\text{Cond}_{\text{REG}}(L)$

$$\forall n \exists w \in L : |w| \geq n \wedge \forall x, z \in \Sigma^*, y \in \Sigma^+ : w = xyz, |xy| \leq n \exists \ell \geq 0 : xy^\ell z \notin L$$

2.2-8 Standardbeispiel

Satz

Die Sprache $\{a^\ell b^\ell : \ell \in \mathbb{N}\}$ ist nicht regulär.

Beweis

- sei n eine beliebige natürliche Zahl
- setze $w = a^n b^n$
- sei $w = xyz$ eine beliebige Zerlegung mit $|xy| \leq n, |y| \geq 1$
- dann gilt $x, y \in \{a\}^*$
- wäre L regulär, so gehörte auch $xy^0z = xz$ zu L („abpumpen“)
- aber das ist nicht der Fall, denn $\#_a x < n$ und $\#_b z = n$ ($\#_\sigma w$ bezeichnet die Anzahl der Vorkommen des Symbols σ in w)

□

Folgerung

2.2-9

Endliche Automaten sind nicht in der Lage, korrekte Klammerungen wie z.B. $((()()))()$ zu überprüfen. Sie scheitern bereits an Eingaben wie diesen:

$\underbrace{(...((()))}_{n} \underbrace{...)}_{n}$

Die Menge D_1 korrekter Klammerungen über einem Klammerpaar $(,)$ ist eine sogenannte *Dycksprache*. Das ist die kleinste Menge, die ε enthält und zu beliebigen enthaltenen Wörtern v, w auch die Wörter vw sowie (v) .

Allgemeine Definition

Die *Dycksprache* D_n ist über einem Alphabet $\{a_1, b_1, \dots, a_n, b_n\}$ der Größe $2n$ definiert⁴:

- $\varepsilon \in D_n$,
- gilt $v, w \in D_n$, so gilt auch $vw \in D_n$ und
- gilt $v \in D_n$, so gilt auch $a_i v b_i \in D_n$ für $i = 1, \dots, n$.

Weitere Wörter sind nicht in D_n enthalten.

Das Pumping-Lemma kann versagen

2.2-10

Beispiel

Jedes nichtleere Wort der Sprache $L = \{a^i b^j c^k : i = 0 \vee j = k\}$ hat einen 1-Pumpfaktor y .

$$y = \begin{cases} a & \text{falls } i \neq 0 \\ b \text{ oder } c & \text{sonst.} \end{cases}$$

Aber

Die Sprache L ist nicht regulär! Formaler Beweis folgt später.

⁴ a_i, b_i bilden ein Klammerpaar

2.3 Automatenminimierung

2.3-1 Beispiel: $(a|b)^*ab$

Ziel: DFA über $\Sigma = \{a, b\}$, der die Eingabe auf das Suffix ab prüft.

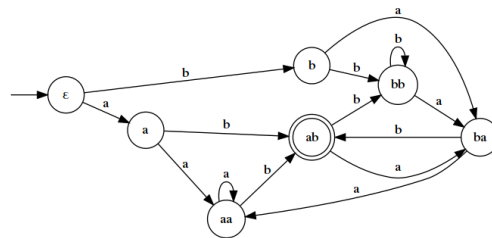
Idee Speichere jeweils letzte beide Zeichen im Zustand, also benutze Zustandsmenge $Q = \{\varepsilon, a, b, aa, ab, ba, bb\}$, mit Startzustand $q_0 = \varepsilon$ und einzigem Endzustand ab .

Übergangstabelle

δ	a	b
ε	a	b
a	aa	ab
b	ba	bb
aa	aa	ab
ba	aa	ab
ab	ba	bb
bb	ba	bb

Die Übergänge sind definiert durch $\delta(q, \sigma) = \text{Suffix von } a\sigma \text{ mit Länge } \min(|qa|, 2)$.

Zustandsdiagramm

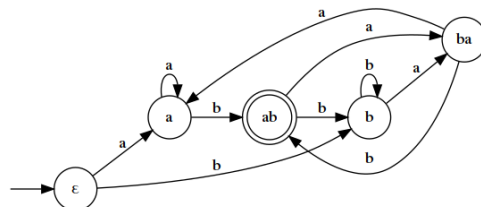


Beobachtung

Ist das letzte Zeichen des Zustands a , dann ist das vorletzte „uninteressant“: der Nachfolgezustand ist unabhängig vom vorletzten Zeichen.

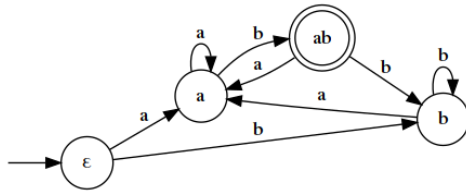
2.3-2 Verschmelzung von Zuständen (1)

Zustände a und aa und auch b und bb haben „gleiche Nachfolger“ (gleiche Folgezustände bei Lesen von a oder b) und werden verschmolzen.



2.3-3 Verschmelzung von Zuständen (2)

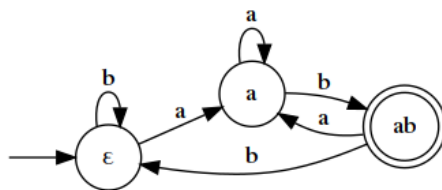
Zustände a und ba haben gleiche Nachfolger und werden verschmolzen.



Verschmelzen von Zuständen (3)

2.3-4

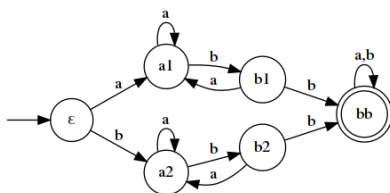
Zustände ε und b haben gleiche Nachfolger und werden verschmolzen.



In diesem Spezialfall leicht einzusehen: Der Automat hat minimale Zustandszahl, um ab -Suffixe zu erkennen.

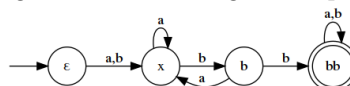
Problem: Die „gleiche Nachfolger“-Idee funktioniert nicht immer!

2.3-5



Dieser Automat hat keine Zustände mit gleichen Nachfolgern, erlaubt dennoch kein „Verschmelzen“. Der folgende erkennt die gleiche Sprache jedoch

bereits mit weniger Zuständen.



Lösung

Gleiche Nachfolgerzustände sind eine unnötig strenge Forderung. Es geht nur um „gleiches Verhalten“ hinsichtlich Akzeptieren/Verwerfen. Erklärung folgt ...

Vorbereitung: Eine Charakterisierung regulärer Sprachen

2.3-6

Definition (Nerode-Relation)

Für eine formale Sprache $L \subseteq \Sigma^*$ heißen $x, y \in \Sigma^*$ **L -äquivalent** (symbolisch $x \equiv_L y$), falls

$$\forall z \in \Sigma^* : xz \in L \iff yz \in L.$$

\equiv_L ist offenbar eine *Äquivalenzrelation*: (1) $\forall x : x \equiv_L x$, (2) $x \equiv_L y \Rightarrow y \equiv_L x$ und (3) $x \equiv_L y \wedge y \equiv_L z \Rightarrow x \equiv_L z$. Σ^* zerfällt also in Äquivalenzklassen bzgl. \equiv_L . $\text{Index}(L)$ bezeichnet die Anzahl $|\Sigma^* / \equiv_L|$ der Äquivalenzklassen.

$x \equiv_L y$ genau dann, wenn für die Linksquotienten gilt $x \setminus L = y \setminus L$ (siehe Paragraph 1.5-5).

Satz (Myhill-Nerode-Theorem)

$L \subseteq \Sigma^*$ ist genau dann regulär, wenn $\text{Index}(L)$ endlich ist.

2.3-7 Beweis, Teil 1: L regulär $\Rightarrow \text{Index}(L) < \infty$

Jeder DFA $D = (Q, \Sigma, \delta, q_0, F)$ mit $L_D = L$ definiert eine Äquivalenzrelation („ D -Äquivalenz“) auf Σ^* :

$$x \equiv_D y \iff \hat{\delta}(q_0, x) = \hat{\delta}(q_0, y).$$

Aus $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$ folgt offenbar auch $\hat{\delta}(q_0, xz) = \hat{\delta}(q_0, yz)$ für jedes $z \in \Sigma^*$.

Somit folgt $xz \in L \iff yz \in L$ (denn D erkennt ja L), d.h. $x \equiv_L y$.

Das bedeutet: Jede Äquivalenzklasse von \equiv_D ist enthalten in einer von \equiv_L . Speziell gilt $|\Sigma^* / \equiv_L| \leq |\Sigma^* / \equiv_D|$.

Rechts steht die Anzahl der von q_0 aus erreichbaren Zustände von D , also ist $\text{Index}(L)$ endlich.

Beobachtung 1

Aus $x \equiv_D y$ folgt $x \equiv_L y$, das bedeutet D -Äquivalenz ist eine **Verfeinerung** (\preceq) von L -Äquivalenz.

2.3-8 Beweis, Teil 2: $\text{Index}(L) < \infty \Rightarrow L$ regulär

Sei $\text{Index}(L)$ endlich. Dann gibt es paarweise nicht- L -äquivalente $x_1, \dots, x_k \in \Sigma^*$, mit

$$\Sigma^* = [x_1] \cup [x_2] \cup \dots \cup [x_k].^5$$

Wir definieren einen DFA D_L^* über Σ mit $Q = \{[x_1], [x_2], \dots, [x_k]\}$ und

$$\delta([x], \sigma) = [x\sigma], q_0 = [\varepsilon], F = \{[x] : x \in L\}.^6$$

Dieser „Äquivalenzklassen-Automat“ D_L^* erkennt genau L : Aus der Definition von δ folgt $\hat{\delta}([\varepsilon], x) = [x]$, also

$$x \in L_{D_L^*} \iff \hat{\delta}(q_0, x) = \hat{\delta}([\varepsilon], x) = [x] \in F \iff x \in L.$$

□

Beobachtung 2: $x \equiv_L y \Rightarrow x \equiv_{D_L^*} y$

Sind x, y L -äquivalent, so ist $[x] = [y]$ und ebenso $[xz] = [yz]$ für alle z . Wegen $\hat{\delta}([x], z) = [xz]$, $\hat{\delta}([y], z) = [yz]$ folgt, dass x, y auch D_L^* -äquivalent sind.

⁵ $[x]$ bezeichnet die Äquivalenzklasse von x

⁶Ist $x \equiv_L x'$, so ist $x\sigma \equiv_L x'\sigma$. Daher ist δ wohldefiniert.

Anwendung 1: „cut & paste“-Argumente

2.3-9

Wenn es zu einer Sprache unendlich viele paarweise nicht- L -äquivalente Wörter gibt, so ist die Sprache nicht regulär.

Die Sprache $L = \{a^n b^n : n \in \mathbb{N}\}$

$x = a^i \not\equiv_L a^j = y$ für $i \neq j$, denn für $z = b^i$ gilt $xz \in L$ aber $yz \notin L$. Also gibt es unendlich viele Äquivalenzklassen $[\varepsilon], [a], [a^2], [a^3], \dots$ bzgl. L — die Sprache ist nicht regulär.

Die „pumpbare“ Sprache $L = \{a^i b^j c^k : i = 0 \vee j = k\}$

$x = ab^j \not\equiv_L ab^\ell = y$ für $j \neq \ell$, denn für $z = c^j$ gilt $xz \in L$ aber $yz \notin L$. Also gibt es unendlich viele Äquivalenzklassen — die Sprache ist nicht regulär.

Anwendung 2: Der „Äquivalenzklassenautomat“ zu $(a|b)^* ab$

2.3-10

Die Nerode-Klassen sind:

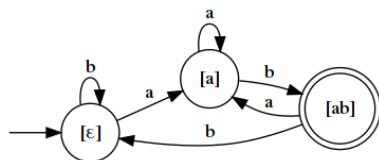
$[ab] = \{x : x \text{ endet mit } ab\} = L.$

$[a] = \{x : x \text{ endet mit } a\},$

$[\varepsilon] = \{x : x \text{ endet nicht auf } a \text{ oder } ab\},$

D_L^* hat Zustandsmenge $Q = \{[\varepsilon], [a], [ab]\}$ mit $q_0 = [\varepsilon], F = \{[ab]\}.$

δ	a	b
$[\varepsilon]$	$[a]$	$[\varepsilon]$
$[a]$	$[a]$	$[ab]$
$[ab]$	$[a]$	$[\varepsilon]$



Der Automat ist **isomorph** zu dem früher betrachteten Automaten aus Paragraph 2.3-4, d.h. bis auf Umbenennung der Zustände gleich.

Die Anzahl der Zustände eines endlichen Automaten A wird seine **Größe** genannt und mit $|A|$ bezeichnet.

Der Minimalautomat

2.3-11

Isomorphiesatz

Ist $L \subseteq \Sigma^*$ regulär, so gilt:

- (1) D_L^* hat minimale Größe unter allen L erkennenden DFAs.
- (2) jeder L erkennende DFA minimaler Größe ist isomorph zu D_L^* .

Beweis

- (1) Ist D ein L erkennender DFA, so ist die Anzahl der Äquivalenzklassen von \equiv_D (und ebenso die Zustandszahl) mindestens so groß ist wie $\text{Index}(L)$. Bei D_L^* ist genau diese Anzahl erreicht, denn Σ^* / \equiv_L und $\Sigma^* / \equiv_{D_L^*}$ sind identisch.
- (2) Falls D genau $\text{Index}(L)$ Zustände hat, so müssen \equiv_L und \equiv_{D_L} identisch sein.

Definition

D_L^* heißt **der Minimalautomat** zu L und wird im weiteren mit $\text{minDFA}(L)$ bezeichnet.

2.3-12 Automatenminimierung

Sei $D = (Q, \Sigma, \delta, q_0, F)$ der zu minimierende DFA. Verschiedene Zustände $q, q' \in Q$ heißen **bisimulativ** (d.h. „verhalten sich gleich“, Notation $q \sim q'$), wenn $\forall z \in \Sigma^* : \hat{\delta}(q, z) \in F \Leftrightarrow \hat{\delta}(q', z) \in F$ Es genügt offenbar, $|z| \leq |Q|$ zu betrachten, ansonsten gäbe es nämlich Zyklen in der Berechnung von $\hat{\delta}(\cdot, z)$ und ungleiches Verhalten ließe sich bereits an einem kürzeren Wort demonstrieren.

Markierungsalgorithmus (oder *table filling*)

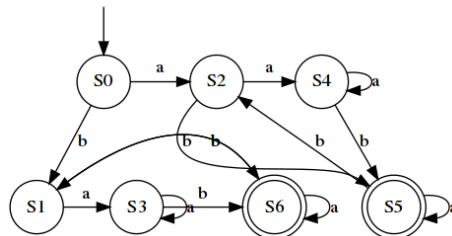
Initialisiere Markiere Paare, die aus einem Endzustand und einem Nicht-Endzustand bestehen (nicht bisimulativ: \nearrow)

Iteriere solange es Änderungen gibt Markiere die unmarkierten Paare, die durch ein Symbol $\sigma \in \Sigma$ in ein markiertes Paar überführt werden (auch nicht bisimulativ: \nearrow)

Terminierte Jetzt noch unmarkierte Paare sind bisimulativ. Verschmelze paarweise deren Zustände.

Bei n Zuständen sind $\frac{n(n-1)}{2} = O(n^2)$ Paare ungleicher Zustände zu betrachten. Für jedes Paar erfordert dies konstant viele Schritte sowie $O(\log_2 n)$ Schritte für das Suchen in der Tabelle — insgesamt also im

2.3-13 wesentlichen quadratischen Aufwand. [Beispiel \(aus +\[Hoffmann](#)



	S1	S2	S3	S4	S5	S6
S0						
S1	-					
S2	-	-				
S3	-	-	-			
S4	-	-	-	-		
S5	-	-	-	-	-	

	S1	S2	S3	S4	S5	S6
S0					✓	✓
S1	-				✓	✓
S2	-	-			✓	✓
S3	-	-	-		✓	✓
S4	-	-	-	-	✓	✓
S5	-	-	-	-	-	

	S1	S2	S3	S4	S5	S6
S0	✓	✓	✓	✓	✓	✓
S1	-				✓	✓
S2	-	-			✓	✓
S3	-	-	-		✓	✓
S4	-	-	-	-	✓	✓
S5	-	-	-	-	-	

Beispiel (Forts.)

2.3-14

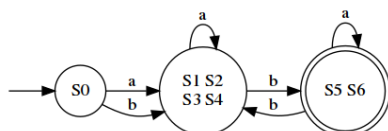
Unmarkierte Paare: $\{S1, S2\}, \{S1, S3\}, \{S1, S4\}$ (und $\{S2, S3\}, \{S2, S4\}, \{S3, S4\}, \{S3, S4\}$) sowie $\{S5, S6\}$

Äquivalenzklassen sind somit

$P1 = \{S0\}, P2 = \{S1, S2, S3, S4\}, P3 = \{S5, S6\}$.

Verschmelzen von Zuständen

Betrachte jede Klasse als Zustand des Minimalautomaten, übernehme Start-/Endzustände und Übergangsfunktion von den Repräsentanten.



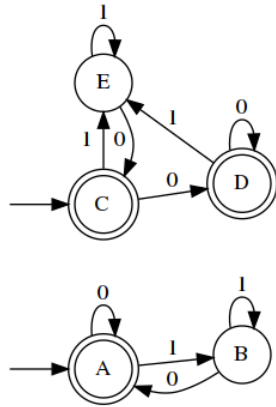
Anwendung 3: DFA-Äquivalenztest

2.3-15

Nach dem Isomorphiesatz (Paragraph 2.3-11) ist der Minimalautomat einer Sprache i.W. eindeutig bestimmt. Damit ist entscheidbar, ob zwei DFAs **sprachäquivalent** sind (die gleiche Sprache erkennen): (1) minimiere beide und

(2) prüfe, ob sie isomorph sind. Wie ist (2) auszuführen? Vom Startzustand ausgehend, systematisch Wege durch die Automaten „abgleichen“ ...?

Idee: Fasse beide Automaten zu einem zusammen und minimiere diesen.



	B	C	D	E
A	✓			✓
B	-	✓	✓	
C	-	-		✓
D	-	-	-	✓

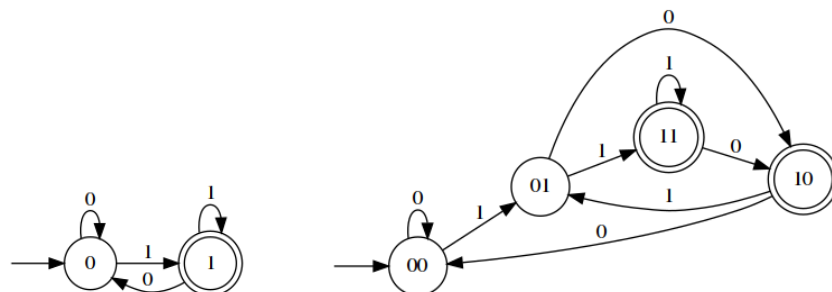
Nach Definition ist die Anzahl der Startzustände für die Bisimulation irrelevant.

2.4 Nichtdeterministische endliche Automaten

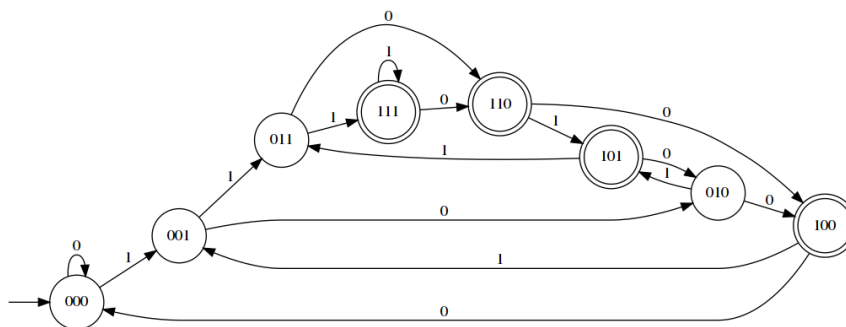
2.4-1 „Einfache“ Sprachen mit großen DFAs

L_{-1} : „letztes Bit = 1“

L_{-2} : „vorletztes Bit = 1“



L_{-3} : „drittletztes Bit = 1“



Exponentielles Wachstum

2.4-2

Bezeichne $\#Q(L)$ die Größe des Minimalautomaten zu L . Bekanntlich gilt $\#Q(L) = \text{Index}(L)$.

Bezeichnung

Für ein Wort $w = w_0w_1\dots w_{n-1} \in \Sigma^*$ und $i \in \{1, 2, \dots, n\}$ bezeichne w_{-i} das „ i -t letzte Zeichen“ von w , d.h.

$$w_{-i} = \begin{cases} w_{n-i} & \text{falls } 1 \leq i \leq n \\ \text{nicht definiert} & \text{sonst.} \end{cases}$$

Behauptung

Für die Sprache $L = L_{-N} := \{w \in \{0, 1\}^* : w_{-N} = 1\}$ gilt $\#Q(L) = 2^N$.

Beweis: Ein „cut & paste-Argument“

2.4-3

2^N Zustände genügen nach der Konstruktionsidee obiger Beispiele:

$Q = \{00\dots 0, \dots, 11\dots 1\}, \delta(b_0\dots b_{n-1}, b_n) = b_1\dots b_n$

1. Betrachte $D = \text{minDFA}(L) = (Q, \{0, 1\}, q_0, \delta, F)$.
2. Wäre $|Q| < 2^N$, dann $\exists x, y \in \{0, 1\}^N, x \neq y$ mit $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$.
3. Die *letzte* Unterscheidungsstelle ist definiert durch $\exists u \in \{0, 1\}^{i-1} : x = \dots 1u, y = \dots 0u$.
4. Betrachte beliebige Verlängerung $v \in \{0, 1\}^{N-i}$.
5. Nach Definition akzeptiert D $xv = \dots \underbrace{1uv}_N$ und verwirft $yv = \dots \underbrace{0uv}_N$.
6. Das widerspricht aber 2., denn $\hat{\delta}(q_0, xv) = \hat{\delta}(q_0, yv)$. □

Definition

2.4-4

Um kompaktere Darstellungen zu ermöglichen kann man den Determinismus aufgeben: anstelle eines Folgezustands tritt eine *Menge von Folgezuständen*, ebenso anstelle des einen Startzustands eine *Menge von Startzuständen*.

Nichtdeterministischer endlicher Automat (NFA)

Ein NFA $N = (Q, \Sigma, \Delta, Q_0, F)$ besteht aus

In vielen Lehrbüchern wird auch für NFA nur ein einziger Startzustand betrachtet. Das ändert aber nichts Grundsätzliches.

- Zustandsmenge Q ,
- Eingabealphabet Σ ,
- Übergangsfunktion $\Delta : Q \times \Sigma \rightarrow 2^Q$ (2^Q bezeichnet die **Potenzmenge** von Q , d.h. die Klasse aller Teilmengen von Q)
- Menge der Startzustände $Q_0 \subseteq Q, Q_0 \neq \emptyset$ und
- Menge F von Endzuständen $F \subseteq Q$.

Bemerkungen

1. Jeder DFA kann als NFA aufgefasst werden (mit Einermengen $\Delta(q, \sigma)$).
2. $\Delta(q, \sigma)$ darf auch leer sein! Äquivalent: Definitionslücke bei (q, σ) , d.h. *Δ ist partiell*.

2.4-5 Erkannte Sprache

Fortsetzung der Übergangsfunktion

Sei N ein NFA mit Übergangsfunktion $\Delta : Q \times \Sigma \rightarrow 2^Q$. Die *fortgesetzte Übergangsfunktion* $\hat{\Delta} : 2^Q \times \Sigma^* \rightarrow 2^Q$ ist wie folgt definiert ($S \subseteq Q$ sei dabei eine Menge von Zuständen): $\hat{\Delta}(S, a) = \bigcup_{q \in S} \Delta(q, a)$ und

$$\hat{\Delta}(S, w) = \begin{cases} S & \text{falls } w = \varepsilon \\ \hat{\Delta}(\Delta(S, a), v) & \text{falls } w = av, a \in \Sigma. \end{cases}$$

- $S \mapsto \Delta(S, a)$ beschreibt die Verarbeitung von a (beginnend mit Zuständen) aus S (*Schrittfront*)
- $S \mapsto \hat{\Delta}(S, w)$ die Verarbeitung von w beginnend in S (*Marsch*)

Definition

NFA $N = (Q, \Sigma, \Delta, Q_0, F)$ **akzeptiert** $w \in \Sigma^*$, falls $\hat{\Delta}(Q_0, w) \cap F \neq \emptyset$ (falls der Marsch beginnend in $\{Q_0\}$ F erreicht).

Die Sprache $L_N = L(N)$ von N ist die Menge der Wörter, die er akzeptiert.

2.4-6 Beispiel

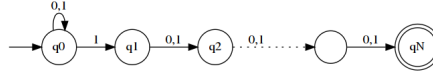
Andere Sichtweise auf die Akzeptierung durch NFA

$w = w_1 w_2 \dots w_n$ wird akzeptiert, falls es $q_0 \in Q_0$ und Zustandsfolge $q_1, q_2, \dots, q_n \in Q$ gibt mit $q_n \in F$ und $\forall i : q_{i+1} \in \Delta(q_i, w_i)$ (*möglicher Lauf*).

M.a.W.: w wird akzeptiert, wenn ein möglicher Lauf F erreicht.

Beispiel: ein NFA für L_N

Der NFA „rät“, auf welche 1 im Eingabestrom es ankommt. Gibt es eine *Möglichkeit* richtig zu raten, wird das Wort akzeptiert.



```
# Beispiel fuer N = 3
{'Q': {'q0', 'q1', 'q2', 'q3'},
 'Sigma': {'0', '1'},
 'Delta': {('q0', '0'): {'q0'},          # Werte sind *Mengen*
            ('q0', '1'): {'q0', 'q1'},   # => es handelt sich um einen
            ('q1', '0'): {'q2'},          # NFA (nicht DFA)
            ('q1', '1'): {'q2'},
            ('q2', '0'): {'q3'}, ('q2', '1'): {'q3'}},
            # Delta('q3', .) ist nicht definiert
 'Q0': {'q0'},
 'F': {'q3'}}
}
```

Bemerkungen

1. Bekanntlich ist auch $\Delta(q, a) = \emptyset$ erlaubt, wie im Beispiel q_N . In der Praxis lässt man diese Wert bequemerweise einfach weg, und verzichtet auf Angaben wie diese:

$(q, 0): \{\}$

2. Ein DFA $D = (Q, \Sigma, q_0, \delta, F)$ kann als NFA $N = (Q, \Sigma, \{q_0\}, \Delta, F)$ aufgefasst werden mit $\Delta(q, a) := \{\delta(q, a)\}$. N ist **äquivalent** zu D d.h. erkennt die gleiche Sprache.

Die Potenzmengenkonstruktion

2.4-7

Wie gesehen: NFAs für L_N kommen mit $N + 1$ Zuständen aus, während DFAs dafür 2^N benötigen — exponentiell mehr!

Erkennen NFAs mehr Sprachen als DFAs? Nein!

Satz von Rabin und Scott (1959)

Zu jedem NFA existiert ein äquivalenter DFA.

Beweis

1. Sei $N = (Q, \Sigma, \Delta, Q_0, F)$ ein NFA.
2. Betrachte $D = (2^Q, \Sigma, \delta, Q_0, F')$ mit

$$\delta(S, a) = \bigcup_{q \in S} \Delta(q, a) \text{ und } F' = \{S \subseteq Q : S \cap F \neq \emptyset\}.$$

3. Beobachte, dass für alle $w \in \Sigma^*$ gilt:

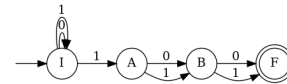
$$\hat{\Delta}(Q_0, w) = \hat{\delta}(Q_0, w).$$

4. Somit akzeptiert N w genau dann, wenn auch D akzeptiert. □

2.4-8 Beispiel

Der NFA für L_{-3}

Zustand	Eingabe	Folgezustandsmengen
I	0	$\{I\}$
I	1	$\{I, A\}$
A	0, 1	$\{B\}$
B	0, 1	$\{F\}$



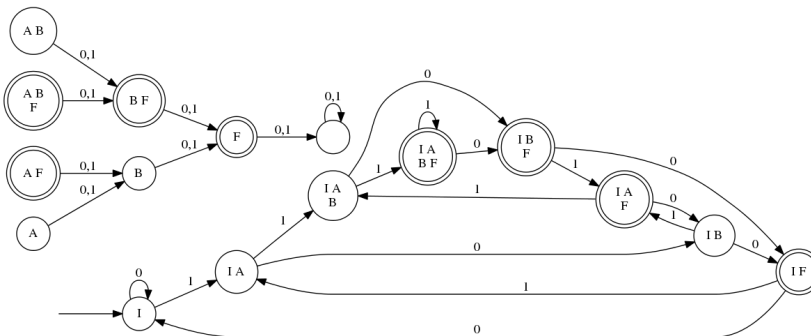
Zugehöriger „Potenzmengenautomat“

Zustand	Eing.	Folgezust.
$\{I\}$	0,1	$\{I\}, \{I, A\}$
$\{I, A\}$	0,1	$\{I, B\}, \{I, A, B\}$
$\{I, B\}$	0,1	$\{I, F\}, \{I, A, F\}$
$\{I, F\}$	0,1	$\{I\}, \{I, A\}$
$\{I, A, B\}$	0,1	$\{I, B, F\}, \{I, A, B, F\}$
$\{I, A, F\}$	0,1	$\{I, B\}, \{I, A, B\}$
$\{I, B, F\}$	0,1	$\{I, F\}, \{I, A, F\}$
$\{I, A, B, F\}$	0,1	$\{I, B, F\}, \{I, A, F\}$

Zustand	Eing.	Folgezust.
$\{\}$	0,1	$\{\}$
$\{A\}$	0,1	$\{B\}$
$\{B\}$	0,1	$\{F\}$
$\{F\}$	0,1	$\{\}$
$\{A, B\}$	0,1	$\{B, F\}$
$\{A, F\}$	0,1	$\{B\}$
$\{B, F\}$	0,1	$\{F\}$
$\{A, B, F\}$	0,1	$\{B, F\}$

Startzustandsmenge $\{I\}$, Endzustandsmenge $F' =$ Klasse der Mengen, die F enthalten

2.4-9 Zustandsdiagramm



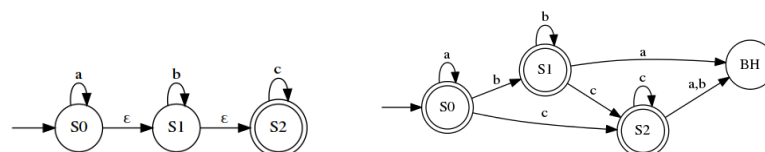
Nicht erreichbare Zustände können weggelassen werden, ohne die erkannte Sprache zu ändern. Es bleibt ein DFA, der zu dem aus dem Beispiel isomorph ist.

2.4-10 ϵ -Übergänge

NFAs werden noch kompakter, wenn Zustandsübergänge ohne Symbol erlaubt sind („spontane Übergänge“). Dies erleichtert auch deutlich den Entwurf.

Beispiel ϵ -NFA und DFA für $\{a^i b^j c^k : i, j, k \geq 0\}$

Beispiel nach: [Hoffmann]



Nichtdeterministischer ε -Akzeptor (ε -NFA)

Ein ε -NFA $N = (Q, \Sigma, \Delta, Q_0, F)$ besteht aus

- Zustandsmenge Q ,
- Eingabealphabet Σ ,
- Übergangsfunktion $\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$,
- Menge der Startzustände $Q_0 \subseteq Q, Q_0 \neq \emptyset$ und
- Menge F von Endzuständen $F \subseteq Q$.

Vorüberlegung zur Fortsetzung von Δ (für die Definition der erkannten Sprache)

2.4-11

Sei $S \subseteq Q$ beliebig.

Ziel: $\hat{\Delta}(S, w)$ soll alle von S aus durch Verarbeiten von $w_1, w_2, \dots, w_{|w|}$ erreichbaren Zustände beinhalten (unterwegs mögliche ε -Übergänge beachten!).

Verarbeitung eines Einzelsymbols

1. unzureichender Ansatz: $\hat{\Delta}(S, a) := \bigcup_{q \in S} \Delta(q, a) \cup \{\Delta(q, \varepsilon) : q \in S\}$
berücksichtigt nur *einzelne* ε -Übergänge und nur *nach* dem Lesen von a
2. erfolgreicher Ansatz: Benutze die ε -Hülle von Zuständen = Menge aller Zustände, die von dort allein durch ε -Übergänge erreicht werden

ε -Hülle eines Zustands q

$$\|q\|_\varepsilon := \{q' : q \xrightarrow{\varepsilon^*} q'\}.$$

$q \xrightarrow{\varepsilon} q'$ steht dabei für $q' \in \Delta(q, \varepsilon)$ und $q \xrightarrow{\varepsilon^*} q'$ für den *reflexiv-transitiven Abschluss* der Relation, also die Existenz einer Kette solcher Beziehungen (eingeschlossen der Fall $q = q'$). Naheliegende Ausdehnung auf Zustandsmengen: $\|S\|_\varepsilon := \bigcup_{q' \in S} \|q'\|_\varepsilon$.

Erinnerung: Ist $R \subseteq M \times M$ eine Relation, so wird ihr **reflexiv-transitiver Abschluss** mit R^* bezeichnet. Elemente $m, m' \in M$ stehen in Relation R^* , falls $m = m'$, oder $\exists m_1 = m, m_2, \dots, m_{k-1}, m_k = m' \in M : m_1 R m_2 \wedge \dots \wedge m_{k-1} R m_k$.

Berechnung der ε -Hülle

Beispiel

Bei dem ε -NFA von Paragraph 2.4-11 gilt: $\|S_0\|_\varepsilon = \{S_0, S_1, S_2\}$,
 $\|S_1\|_\varepsilon = \{S_1, S_2\}$, $\|S_2\|_\varepsilon = \{S_2\}$

Sättigungsalgorithmus

```
def eclosure(E,S):      # E = eps-NFA, S = Menge von Zuständen
    (Delta,changed) = (E['Delta'],True)
    while changed:
        changed = False
        T = S
        for s in S:
            if (s,'') in Delta.keys(): # ist dieser eps-Uebergang definiert?
                S = S.union(Delta[(s,'')])
            if (S != T): changed = True
    return S
```

Da Q endlich ist, terminiert der Algorithmus.

2.4-13 Simulation durch DFA

Modifizierte Fortsetzung von Δ

Schrittfront $\hat{\Delta}'(S, a) = \parallel \underbrace{\hat{\Delta}(\parallel S \parallel_\varepsilon, a)}_{\text{fortgesetzte \ddot{U}F des NFA}} \parallel_\varepsilon$ für $a \in \Sigma$ und

Marsch

$$\hat{\Delta}'(S, w) = \begin{cases} \parallel S \parallel_\varepsilon & \text{falls } w = \varepsilon \\ \hat{\Delta}'(\hat{\Delta}'(S, a), v) & \text{falls } w = av, a \in \Sigma. \end{cases}$$

Erkannte Sprache

w wird von ε -NFA $N = (Q, \Sigma, \Delta, Q_0, F)$ *akzeptiert*, falls $\hat{\Delta}'(Q_0, w) \cap F \neq \emptyset$.

Die von N *erkannte Sprache* L_N ist die Menge der von N akzeptierten Wörter.

Satz (ε -Reduktionstheorem)

Jeder ε -NFA kann durch einen DFA simuliert werden.

Beweisidee

Anpassung der Potenzautomatenkonstruktion auf diesen Fall. Für ε -NFA $N = (Q, \Sigma, Q_0, \Delta, F)$ betrachte $D = (2^Q, \Sigma, Q'_0, \delta, F')$ mit $Q'_0 = \parallel Q_0 \parallel_\varepsilon$, $\delta(S, a) = \parallel \bigcup_{q \in S} \Delta(q, a) \parallel_\varepsilon$ und $F' = \{S \subseteq Q : S \cap F \neq \emptyset\}$.

Beobachte, dass für alle $w \in \Sigma^*$ gilt: $\hat{\Delta}(Q_0, w) = \hat{\delta}(Q'_0, w)$.

Somit akzeptiert N w genau dann, wenn auch D akzeptiert. □

Beispiel

siehe Jupyter-Notebook

siehe Jupyter-Notebook

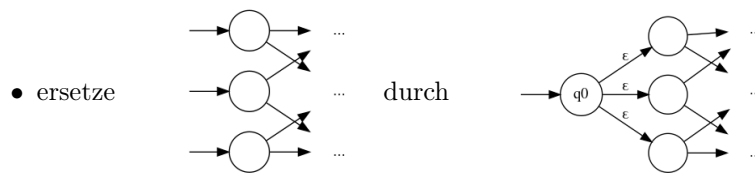
2.4-14 Normierte ε -NFAs

Meist werden NFAs mit *nur einem* anstelle einer *Menge* von Startzuständen wie die Beispiele in Paragraph 2.4-1 definiert. Der Unterschied ist unwesentlich, denn beide Varianten erkennen die gleiche Klasse von Sprachen:

Vom Startzustand zur Startmenge

- ersetze $(Q, \Sigma, \Delta, q_0, F)$ durch $(Q, \Sigma, \Delta, \{q_0\}, F)$

Von Startmenge zum Startzustand



Satz

Für jede reguläre Sprache L gibt es einen erkennenden ε -NFA mit einem einzigen Startzustand und einem einzigen Endzustand (**normierter ε -NFA**).

Beweis: analog

2.5 Abschlusseigenschaften

Abschlusseigenschaften von $\text{REG}(\Sigma)$

2.5-1

Satz

Sei Σ ein Alphabet und seien $L = L_1, L_2 \subseteq \Sigma^*$ reguläre Sprachen. Dann sind folgende Sprachen ebenfalls regulär:

1. die Vereinigung $L_1 \cup L_2$, Produkt $L_1 L_2$ und Kleene-Stern L^*
2. das Komplement \overline{L} und die Reflexion L^R
3. der Schnitt $L_1 \cap L_2$
4. das Bild $h(L)$ bzw. Urbild $g^{-1}(L)$, unter Homomorphismen $h : \Sigma^* \rightarrow \Gamma^*$ bzw. $g : \Gamma^* \rightarrow \Sigma^*$
5. Quotient L_1/L_2 und Shuffle-Produkt $L_1 \# L_2$

Beweisideen

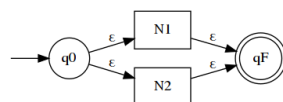
- Modifikation/Kombination von Akzeptoren für L_1, L_2 zu Akzeptor für die Zielsprache
- Manipulation regulärer Ausdrücke

Abschluss unter Vereinigung, Produkt und Stern (Thompson-Konstruktion)

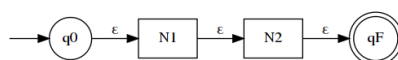
2.5-2

Wir betrachten *normierte ε -NFAs* $N1, N2$ für $L = L_1, L_2$:

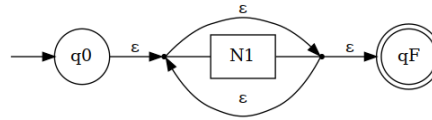
Vereinigung



Produkt



Kleene-Stern



2.5-3 Intermezzo: vom regulären Ausdruck zum NFA

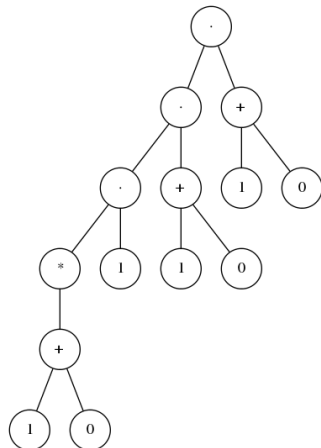
Es ist einfach, zu jedem der regulären Basisausdrücke \emptyset , $\{\varepsilon\}$ und $\{\sigma\}$, $\sigma \in \Sigma$ einen DFA anzugeben, der die beschriebene Wortmenge erkennt. Die Thompson-Konstruktion zeigt, dass *jede* durch einen regulären Ausdruck beschriebene reguläre Menge durch einen endlichen Automaten erkannt wird.

Wir demonstrieren die Konstruktion anhand des regulären Ausdrucks

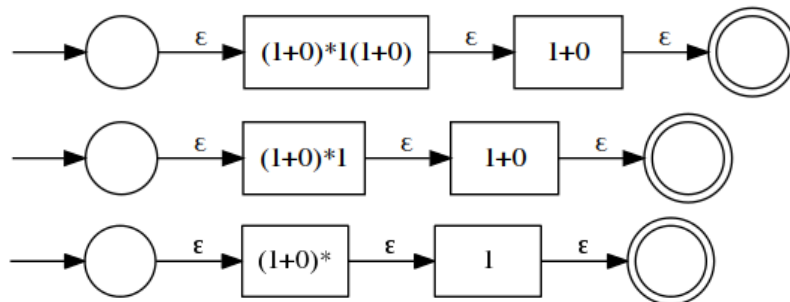
$$(1 + 0) * 1(1 + 0)(1 + 0)$$

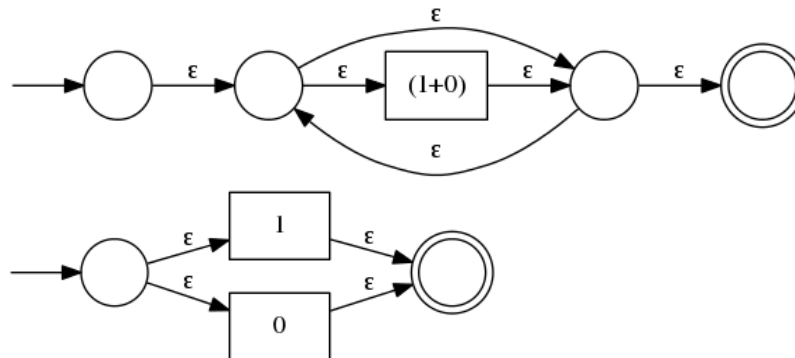
2.5-4 $(1 + 0) * 1(1 + 0)(1 + 0)$

Strukturbaum



Konstruktionsschritte (nur linker Pfad)





Automatisierung (Dijkstras Shunting-yard-Algorithmus)

Aus technischen Gründen wird der Ausdruck mit vollständiger Klammerung und explizit angegebener Konkatenation (\cdot) betrachtet:

$$(((1 + 0) * \cdot 1) \cdot (1 + 0)) \cdot (1 + 0))$$

Idee: Anstelle des Strukturbaums (implizit) die **Postfix**-Schreibweise des Ausdrucks verwenden, d.h. Operatoren *nach* Operanden:

$$(((1 + 0) * \cdot 1) \cdot (1 + 0)) \cdot (1 + 0)) \quad \mapsto \quad 10 + * 1 \cdot 10 + \cdot 10 + \cdot$$

Prinzip: Verarbeite geklammerte Teilausdrücke durch NFA-Kombination, also beim zeichenweisen Lesen des Ausdrucks:

- Symbole $\in \Sigma = \{0, 1\}$ und Zwischenergebnisse in *Wertestack* speichern, Operatorzeichen $\in \{+, \cdot, *\}$ in *Operatorstack* (Klammern steuern nur den Ablauf \Rightarrow nicht speichern)
- wenn Teilausdruck beendet, dessen Operator und Operanden beschaffen und verarbeiten, d.h.
 - bei Symbolen $\in \Sigma = \{0, 1\}$: `mk_symbol_nfa` aufrufen, sonst
 - je nach Operator `mk_plus_nfa`, `mk_cat_nfa` oder `mk_star_nfa`
 dann Ergebnis in Wertestack speichern.

Vom r.A. zum NFA: Vereinfachte Eigenimplementation

```
# Eingabe: r.A. als String (Bsp. "((c*)+(a.b))" anstelle "ab+c*", d.h. ...
def RE2NFA(re): # .. vollstaendig geklammert und explizit konkateniert)
    ops = []; values = [] # Listen (werden als Stack-Speicher benutzt)
    for s in re:
        # Ausdruck zeichenweise lesen
        if s == '(': pass # Anfang von Teilausdruck
        elif s in ['+', '.', '*']: ops.append(s) # entspricht: ops.push(..)
        elif s == ')': # Ende v. Teilausdruck, Verarbeitung:
            op = ops.pop()
            if op == '+':
                (fa,fb) = (values.pop(), values.pop())
                values.append(mk_plus_nfa(fb,fa)) # Parallelschaltung
            elif op == '.':
                (fa,fb) = (values.pop(), values.pop())
                values.append(mk_cat_nfa(fb,fa)) # Reihenschaltung
            elif op == '*':
                fa = values.pop()
                values.append(mk_star_nfa(fa)) # Schleifenkonstruktion
```

```

else: values.append(mk_symbol_nfa(s))
return(values.pop())

```

Besser: Die Jove-Implementation re2nfa

Die Implementation beruht auf `lex` und ist natürlich vorzuziehen:

- Konkatenation nicht explizit angeben (d.h. `(ab)` anstelle `(a.b)`)
- Assoziativität und Prioritäten werden berücksichtigt z.B. sind `((c*)+(ab))+d` und `c*+ab+d` gleichwertig

siehe Jupyter-Notebook

2.5-7 Satz von Kleene

Folgender Satz zeigt, dass die Unterscheidung zwischen regulären Mengen (Kapitel 1) und regulären Sprachen tatsächlich nicht nötig ist.

Satz (Kleene 1956)

Für jeden DFA $D = (Q, \Sigma, \delta, q_0, F)$ gibt es einen regulären Ausdruck γ über Σ mit $M_\gamma = L(D)$.

Beweisidee

- für jedes $q \in F$ betrachte $L_D(q) := \{w : \hat{\delta}(q_0, w) = q\}$
- konstruiere dafür einen regulären Ausdruck: induktiv durch schrittweise Hinzunahme erlaubter „Zwischenzustände“ auf dem Weg $q_0 \rightsquigarrow q$
- beschreibe Vereinigung all dieser Sprachen durch regulären Ausdruck

2.5-8 Kleenes Algorithmus (Idee)

$z_i \xrightarrow{w,k} z_j$ steht für: w führt von z_i zu z_j , mit Zwischenzuständen mit Index $\leq k$

Sei $Q = \{z_1, \dots, z_n\}$ mit $q_0 = z_1$. Offenbar $L(D) = \bigcup_{z_i \in F} R_{1,i}^n$, wobei

$$R_{i,j}^k = \{w \in \Sigma^* \mid z_i \xrightarrow{w,k} z_j\}$$

Nun gilt:

- $R_{i,i}^0 = \{\varepsilon\} \cup \{a \in \Sigma \mid \delta(z_i, a) = z_i\}$, $i \neq j \Rightarrow R_{i,j}^0 = \{a \in \Sigma \mid \delta(z_i, a) = z_j\}$, und
- $R_{i,j}^{k+1} = R_{i,j}^k \cup R_{i,k+1}^k (R_{k+1,k+1}^k)^* R_{k+1,j}^k$

Konstruiere entsprechend rekursiv zu jedem $R_{i,j}^k$ einen äquivalenten r.A. $\gamma_{i,j}^k$.

Ist $F = \{z_{i_1}, \dots, z_{i_m}\}$, dann beschreibt der reguläre Ausdruck

$$\gamma = \gamma_{1,i_1}^n \mid \gamma_{1,i_2}^n \mid \dots \mid \gamma_{1,i_m}^n$$

genau die von D erkannte Sprache. \square [Abschlusseigenschaften: Komplement und Reflexion](#)

2.5-9

Sei $L \in \text{REG}(\Sigma)$ und $D = (Q, \Sigma, \delta, q_0, F)$ ein DFA mit $L = L_D$.

Komplement: Finalmenge komplementieren (Jove: comp_dfa)

Für $\overline{D} := (Q, \Sigma, \delta, q_0, Q \setminus F)$ gilt:

$$w \in L_{\overline{D}} \Leftrightarrow \hat{\delta}(q_0, w) \in Q \setminus F \Leftrightarrow \hat{\delta}(q_0, w) \notin F \Leftrightarrow w \notin L_D \Leftrightarrow w \in \overline{L}$$

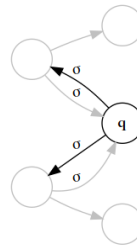
Reflexion: Start- und Finalzustände austauschen (Jove: rev_dfa)

Für den NFA (nicht DFA!) $D^R := (Q, \Sigma, \Delta, F, \{q_0\})$ mit

$$\forall (q, \sigma) \in Q \times \Sigma : \Delta(q, \sigma) = \{q' \in Q : \delta(q', \sigma) = q\}$$

gilt $w \in L_D$ genau dann

- wenn der w -Lauf von q_0 zu einem Endzustand führt
- wenn „ein w^R -Lauf von einem $q \in F$ zu q_0 führt“
- wenn $w^R \in L_{D^R}$



Weitere Abschlusseigenschaften

2.5-10

Schnitt: umformen mit Komplement und Vereinigung

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Alternative: **Produktautomat** Gegeben DFAs

$D_i = (Q_i, \Sigma, \delta_i, q_{0i}, F_i), i \in \{1, 2\}$, so bezeichne

$D_1 \times D_2 := (Q_1 \times Q_2, \Sigma, \delta, q_0 = (q_{01}, q_{02}), F_1 \times F_2)$ den Automaten mit $\delta((q_1, q_2), \sigma) = (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma))$ für alle $q_1 \in Q_1, q_2 \in Q_2$. Wenn D_1, D_2 gerade L_1, L_2 erkennen, erkennt $D_1 \times D_2$ offensichtlich genau $L_1 \cap L_2$.

Quotient und Shuffle-Produkt: Übungsaufgabe

Homomorphes Bild: siehe Kapitel 1

siehe Jupyter-Notebook

Homomorphes Urbild:

Sei $D = (Q, \Gamma, \delta, q_0, F)$ ein DFA für $L \subseteq \Gamma^*$ und $g : \Sigma^* \rightarrow \Gamma^*$ ein Homomorphismus. Das Urbild $g^{-1}(L) \subseteq \Sigma^*$ wird erkannt durch den DFA $D' = (Q, \Sigma, \delta', q_0, F)$ mit

$$\forall (q, \sigma) \in Q \times \Sigma : \delta'(q, \sigma) = \hat{\delta}(q, g(\sigma)).$$

Begründung: $\hat{\delta}'(q_0, \sigma_1 \dots \sigma_n) = \hat{\delta}(q_0, g(\sigma_1 \dots \sigma_n))$.

□

2.6 Reguläre Grammatiken

Ersetzungssysteme

2.6-1

Idee

Wörter einer formalen Sprache $\subseteq \Sigma^*$ durch Ersetzung generieren:

- starte mit einer *Variablen* $\notin \Sigma$ (Variablen sind Platzhalterzeichen)
- wende gewisse *Produktionen* an von der Form

linke Seite \rightarrow rechte Seite

bis alle Variablen durch *Terminalzeichen* $\in \Sigma$ ersetzt sind

Beispiel: Natürliche Sprache

Wenn man z.B. Sätze einer natürlichen Sprache als Wörter einer formalen Sprache betrachtet, sind grammatische Kategorien Platzhalter für das, was dann im Text erscheint: Wörter, Satzzeichen, ...

$\langle \text{Satz} \rangle \rightarrow \langle \text{Subjekt} \rangle \langle \text{Praedikat} \rangle \langle \text{Objekt} \rangle.$
 $\langle \text{Subjekt} \rangle \rightarrow \langle \text{Artikel} \rangle \langle \text{Adjektiv} \rangle \langle \text{Substantiv} \rangle$
 $\langle \text{Artikel} \rangle \rightarrow \text{Der} \mid \text{Die} \mid \text{Das} \mid \dots$
 $\langle \text{Adjektiv} \rangle \rightarrow \text{kleine} \mid \text{große} \mid \dots$
 $\langle \text{Substantiv} \rangle \rightarrow \text{Eisbär} \mid \text{Dachs} \mid \text{Nilpferd} \mid \dots$
 $\langle \text{Praedikat} \rangle \rightarrow \text{mag} \mid \text{hasst} \mid \dots$
 $\langle \text{Objekt} \rangle \rightarrow \text{Kekse} \mid \text{Torte} \mid \dots$

Beispiel: Python

<https://docs.python.org/3/reference/grammar.html>

2.6-2 Chomsky-Grammatiken

Definition

Eine **Chomsky-Grammatik** (oder einfach Grammatik) $G = (V, \Sigma, P, S)$ besteht aus

- einer endlichen Menge V (**Variablen**),
- einem Alphabet Σ (**Terminalalphabet**) mit $V \cap \Sigma = \emptyset$,
- einer endlichen Menge $P \subseteq (V \cup \Sigma)^+ \times (V \cup \Sigma)^*$ von **Produktionen** und
- einem **Startsymbol** $S \in V$.

Erinnerung: A^+ bezeichnet die Menge der nichtleeren Wörter über A (vgl. Paragraph 1.3-5)

Gängige Terminologie und Konventionen

- *Variablen* = *Nonterminale*
werden meist mit Großbuchstaben bezeichnet: S, A, B_1, \dots ,
Terminale mit Kleinbuchstaben $a, b, \dots, 0, 1, \dots$
- *Produktionen* ($\text{linkeSeite}, \text{rechteSeite}$) $\in P$ werden meist als (Ersetzungs-) **Regeln** notiert: $\text{linkeSeite} \rightarrow \text{rechteSeite}$
- **Satzformen** von G = Wörter aus $(V \cup \Sigma)^*$
werden häufig mit griechischen Buchstaben bezeichnet: α, β, \dots

Ableitungsrelation

In G ist u **direkt ableitbar** zu v (symbolisch: $u \Rightarrow_G v$), falls es Satzformen x, z und eine Regel $(y \rightarrow y') \in P$ gibt mit $u = xyz$ und $v = xy'z$.

Allgemeiner: u ist **ableitbar** zu v , falls es Satzformen w_0, w_1, \dots, w_n gibt, mit

$$u = w_0 \Rightarrow_G w_1 \wedge w_1 \Rightarrow_G w_2 \wedge \dots \wedge w_{n-1} \Rightarrow_G w_n = v.$$

Symbolisch ausgedrückt: $u \Rightarrow_G^* v$ (reflexiv-transitive Hülle von \Rightarrow_G).

Insbesondere gilt $u \Rightarrow_G u$.

Wenn klar (oder irrelevant) ist, was G ist, kann man auch einfach \Rightarrow bzw. \Rightarrow^* schreiben.

Die von der Grammatik erzeugte Sprache

$$L(G) = \{w \in \Sigma^* : S \Rightarrow_G^* w\}.$$

Eine Folge $S \Rightarrow_G \dots \Rightarrow_G w \in L(G)$ heißt **Ableitung** von w .

$L(G)$ wird auch kurz „die Sprache von G “ genannt

Ein Beispiel

Die Grammatik $G = (\{E, T, F\}, \{(\cdot), a, +, \cdot\}, P, E)$ mit Startsymbol E und P bestehend aus diesen Regeln

$$\begin{array}{ll} E & \rightarrow T \\ E & \rightarrow E + T \\ T & \rightarrow F \\ T & \rightarrow T \cdot F \\ F & \rightarrow a \\ F & \rightarrow (E) \end{array}$$

generiert korrekt geklammerte arithmetische Ausdrücke, wie $a \cdot a \cdot (a + a) + a$. Zum Beispiel durch fortlaufende Ersetzung des am weitesten links bzw. rechts stehenden Nonterminals (**Linksableitung** bzw. **Rechtsableitung**):

$$E \Rightarrow E + T \Rightarrow T + T \Rightarrow T \cdot F + T \Rightarrow \dots \Rightarrow a \cdot a \cdot (a + a) + F \Rightarrow a \cdot a \cdot (a + a) + a$$

$$E \Rightarrow E + T \Rightarrow E + F \Rightarrow E + a \Rightarrow \dots \Rightarrow T \cdot a \cdot (a + a) + a \Rightarrow a \cdot a \cdot (a + a) + a.$$

Es sind weitere möglich. Wichtig ist nur, dass es *irgendeine* Ableitung gibt.

Reguläre Grammatiken

2.6-5

Die Menge der korrekt geklammerten arithmetischen Ausdrücke ist *keine* reguläre Sprache. Das folgt aus den Abschlusseigenschaften regulärer Sprachen, denn der Homomorphismus der gegeben ist durch

$$h : x \mapsto \begin{cases} x & \text{falls } x \in \{(\cdot)\} \\ \varepsilon & \text{sonst} \end{cases}$$

bildet die Menge auf eine Dyck-Sprache (siehe Paragraph 2.2-9) ab, die bekanntlich nicht-regulär ist. Aber es gibt Grammatiken, die reguläre Sprachen erzeugen.

Definition

G heißt **reguläre** oder **Typ3**-Grammatik, falls

$$P \subset V \times (\Sigma \cup \Sigma V).$$

2.6-6 Beispiel einer Typ3-Grammatik

Die Grammatik $G_1 = (\{T, M\}, \{a, \cdot\}, P, T)$ mit Regeln

$P = \{T \rightarrow a, T \rightarrow aM, M \rightarrow \cdot T\}$ erzeugt die Menge aller ungeklammerten Multiplikationsterme $\{a, a \cdot a, a \cdot a \cdot a, \dots\} = L(a(\cdot a)^*)$.

Ableitung von $a \cdot a \cdot a$

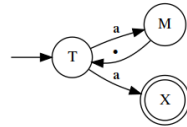
$$T \Rightarrow aM \Rightarrow a \cdot T \Rightarrow a \cdot aM \Rightarrow a \cdot a \cdot T \Rightarrow a \cdot a \cdot a$$

Diese (einzig mögliche) Ableitung erzeugt die Terminale von links nach rechts. Vergleich: Ein NFA konsumiert die Eingabe von links nach rechts.

2.6-7 Von der Grammatik zum NFA

Äquivalenter NFA

Beispiel:



$\dot{\cup}$ steht für disjunkte Vereinigung,
d.h. X ist verschieden von allen
Nonterminalen

Und allgemein:

Typ3-Grammatik $G = (V, \Sigma, P, S) \mapsto$ NFA $N_G = (Q, \Sigma, \Delta, Q_0, F)$ mit

- $Q = V \dot{\cup} \{X\}$
- $Q_0 = \{S\}$
- $F = \{X\}$
- $\Delta(A, \sigma) \ni B$ falls $A \rightarrow \sigma B \in P$ und
 $\Delta(A, \sigma) \ni X$ falls $A \rightarrow \sigma \in P$

Korrektheit

Offensichtlich wird das leere Wort weder von G generiert noch von N_G akzeptiert. Sei also $w \in \Sigma^*$ nichtleer. Dann gilt

$$\begin{aligned} w_1 w_2 \dots w_n \in L(G) &\Leftrightarrow \exists A_1, \dots, A_{n-1} \in V : S \Rightarrow w_1 A_1 \Rightarrow w_1 w_2 A_2 \Rightarrow \dots \Rightarrow w_1 w_2 \dots w_n \\ &\Leftrightarrow \exists A_1, \dots, A_{n-1} \in Q : \Delta(S, w_1) \ni A_1, \dots, \Delta(A_{n-1}, w_n) \ni X \\ &\Leftrightarrow w_1 w_2 \dots w_n \in L(N_G) \quad \square \end{aligned}$$

2.6-8 ε -Produktionen

Wir erlauben zusätzlich bei Typ3-Grammatiken die Regel $S \rightarrow \varepsilon$, wodurch $\varepsilon \in L(G)$ möglich wird.

Beispiel

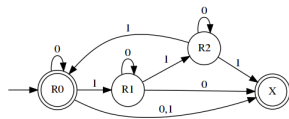
Die Typ3-Grammatik mit Startsymbol R_0 und den Regeln

$$\begin{array}{lll}
R_0 & \rightarrow & \varepsilon \\
R_0 & \rightarrow & 0R_0 \\
R_0 & \rightarrow & 1R_1 \\
R_0 & \rightarrow & 0 \\
R_0 & \rightarrow & 1 \\
R_1 & \rightarrow & 0R_1 \\
R_1 & \rightarrow & 1R_2 \\
R_1 & \rightarrow & 0 \\
R_2 & \rightarrow & 0R_2 \\
R_2 & \rightarrow & 1R_0 \\
R_2 & \rightarrow & 1
\end{array}$$

erzeugt die Sprache $\{w \in \{0,1\}^* : \#_1 w \pmod 3 \in \{0,1\}\}$.

Äquivalenter NFA

Beispiel:



Allgemein: Falls $S \rightarrow \varepsilon \in P$ wird der Automat prinzipiell so konstruiert wie zuvor, nur dies geändert:

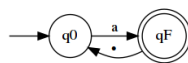
- $F = \{S, X\}$ (anstelle $F = \{X\}$)

Vom DFA zur Grammatik

2.6-9

Ist $D = (Q, \Sigma, \delta, q_0, F)$ ein DFA, so ist folgende Typ3-Grammatik äquivalent dazu: $G_D = (V, \Sigma, P, S)$ mit $V = Q$, $S = q_0$ und $P = \{q \rightarrow \sigma q' : \delta(q, \sigma) = q'\} \cup \{q \rightarrow \sigma : \delta(q, \sigma) \in F\}$ (im Falle $q_0 \in F$ wird zusätzlich $q_0 \rightarrow \varepsilon$ aufgenommen)

Beispiel



$$\begin{array}{ll}
q_0 & \rightarrow aq_F \\
q_0 & \rightarrow a \\
q_F & \rightarrow \cdot q_0
\end{array}$$

D akzeptiert genau dann das leere Wort, wenn $q_0 \in F$ und genau dann wird es von G_D generiert. Sei also $w \in \Sigma^*$ nichtleer. Dann gilt

$$\begin{aligned}
w_1 w_2 \dots w_n \in L(D) &\Leftrightarrow \exists q_1, \dots, q_n \forall i, 0 \leq i < n : \delta(q_i, w_{i+1}) = q_{i+1} \wedge q_n \in F \\
&\Leftrightarrow \exists q_1, \dots, q_n \forall i, 0 \leq i < n - 1 : q_i \rightarrow w_{i+1} q_{i+1} \in P \wedge q_{n-1} \rightarrow w_n \in P \\
&\Leftrightarrow w_1 w_2 \dots w_n \in L(G_D) \quad \square
\end{aligned}$$

Beobachtung

Das wird später noch wichtig: Da ein akzeptiertes Wort nur einen „Lauf durch den Automaten D “ hat, ist die Ableitung dieses Worts in der zugeordneten Grammatik *eindeutig bestimmt*.

siehe Jupyter-Notebook
2.6-10

Typ3-Sprachen

Eine formale Sprache $L \subset \Sigma^*$ heißt „vom Typ 3“ oder **Typ3-Sprache**, falls es eine Typ3-Grammatik G gibt mit $L(G) = L$.

Insgesamt haben wir gezeigt:

Satz

Sei Σ ein Alphabet. Die Menge der Typ3-Sprachen über Σ ist gleich der Menge der regulären Sprachen.

2.6-11 Zwischenbilanz

Wir kennen drei verschiedene Konzepte zur Charakterisierung regulärer Sprachen:

- endliche Automaten *erkennen* reguläre Sprachen,
- reguläre Ausdrücke *beschreiben* reguläre Sprachen und
- reguläre Grammatiken *generieren* reguläre Sprachen.

Theoretischer Nutzen

Der Begriff der regulären Sprachen ist keine ad hoc Begriffsbildung. Unterschiedliche Konzepte erlauben sehr flexible Argumentation.

Praktischer Nutzen

Interessante Anwendungsmöglichkeiten, z.B. Syntaxprüfung mit „Zertifikat“:

Problem Passt Eingabe w zu gegebenem regulären Ausdruck r ?

Lösung

1. äquivalenten NFA N konstruieren (Thompson),
2. NFA in äquivalenten minimalen DFA D umwandeln,
3. Lauf von w in D entspricht Ableitung in Grammatik,
4. Ableitung „zertifiziert“, dass $w \in L(r)$.

2.7 String Matching mit endlichen Automaten

2.7-1 Das String-Matching-Problem

Gegeben $m, t \in \Sigma^*$

Entscheide Ist das *Muster* m Teilwort von *Text* t ?

Die Texte, die m enthalten, bilden die reguläre Sprache $L_m := \Sigma^* m \Sigma^*$. Ein geeigneter DFA $D_m = (Q, \Sigma, \delta, q_0, F)$ kann entscheiden, ob $t \in L_m$ gilt.

Vorüberlegung

Präfixe von m als „Fortschrittsanzeige“ nutzen, wie in diesem Beispiel:

- Muster: $abab$, Text: $abbababaa\dots$
- Zustandsfolge: $\varepsilon, a, ab, \varepsilon, a, ab, aba, abab$ — Akzeptierung

Also

- Zustände = Präfixe von m
- aktueller Zustand = längstes Präfix von m , mit dem der bereits gelesene Teil der Eingabe endet (anfangs: ε).

Somit: $Q = \text{Pref}(m) := \{p : p \text{ ist Präfix von } m\}, q_0 = \varepsilon, F = \{m\}$

Die Übergangsfunktion

2.7-2

Vorbereitung: Die „längste Präfix“-Funktion von m

- Definiere $\ell_m : \Sigma^* \rightarrow \text{Pref}(m)$ durch
 $\ell_m(u) :=$ längstes Präfix von m mit dem u endet
 Beispiel: für $m = abab$ ist $\ell_m(babab) = abab$ und $\ell_m(bababa) = aba$
- Also: $\ell_m(u) = p \Rightarrow \ell_m(u\sigma) = \ell_m(p\sigma)$. M.a.W.: Die Funktionswerte von ℓ_m sind bereits durch deren Werte auf Präfixen von m festgelegt.

Definition der Übergangsfunktion

$$\delta(p, \sigma) = \begin{cases} \ell_m(p\sigma) & , \text{ für } p \neq m \\ m & , \text{ für } p = m. \end{cases}$$

Der Minimalautomat

2.7-3

Satz

$D_m = (Q, \Sigma, \delta, q_0, F)$ ist der Minimalautomat für die Sprache L_m .

Teil 1: D_m erkennt genau L_m

Äquivalent: Für jedes Wort $x \in \Sigma^*$ gilt

ENTWEDER $\hat{\delta}(\varepsilon, x) = m$ (und m ist Teilwort von x) ODER $\ell_m(x) \neq m$

Induktionsanfang Aussage gilt offenbar für $x = \varepsilon$

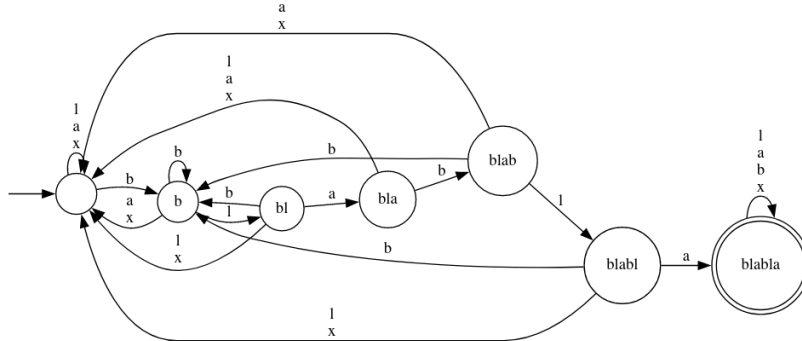
Induktionsschritt Sei $x' = x\sigma$. Wir betrachten zwei Fälle:

1. Ist $\hat{\delta}(\varepsilon, x) = m$, dann ist m Teilwort von x und auch von $x\sigma$ und somit $\hat{\delta}(\varepsilon, x') = \hat{\delta}(\varepsilon, x\sigma) = \delta(m, \sigma) = m$
2. Anderenfalls $\hat{\delta}(\varepsilon, x) \neq m$, dann
 $\hat{\delta}(\varepsilon, x') = \hat{\delta}(\varepsilon, x\sigma) = \delta(\hat{\delta}(\varepsilon, x), \sigma) = \delta(\ell_m(x), \sigma) = \ell_m(x\sigma)$

Teil 2: D_m hat minimale Größe.

- Das kürzeste Wort in $\Sigma^*m\Sigma^*$ ist m selbst.
- Angenommen $|m|$ Zustände genügen, so wird bei Eingabe $t = m$ ein Zustand wiederholt, d.h. auch ein Wort kürzerer Länge wird akzeptiert (wie beim Pumping Lemma). Widerspruch!

2.7-4 Beispiel: blabla



x steht stellvertretend für ein beliebiges Symbol außer **a**, **b**, **1**.

2.8 Entscheidungsprobleme für reguläre Sprachen

2.8-1 Das Wortproblem

Sei $L \subseteq \Sigma^*$ eine reguläre Sprache und $w \in \Sigma^*$. Gilt $w \in L$?

Das ist entscheidbar durch Simulation eines entsprechenden endlichen Automaten.

Beispiel: NFA A auf $w \in \Sigma^N$ simulieren

(siehe `jove/Def_NFA.py`, hier vereinfacht wiedergegeben)

```
def step_nfa(A,S,sigma):# welche Zustaende erreicht die von S ausgehende ..
    T = set()           # .. Schrittfrent "in Richtung sigma"?
    for q in S: T = T|A['Delta'][(q,sigma)] # Zeichen | bedeutet Vereinigung
    return T
def run_nfa(A, S, w):    # welche Zustaende erreicht der von S ausgehende ..
    if w=="": return S  # .. "Marsch entlang w"?
    else: return run_nfa(A, step_nfa(A, S, w[0]), w[1:])
def accepts_nfa(A, w):   # wird wenigstens ein F-Zustand erreicht?
    # ...
```

Bei ℓ Zuständen ist die Laufzeit $O(\ell^2 \log \ell \cdot N)$ (bzw. $O(N \cdot \log \ell)$ bei DFAs). Der Faktor $\log \ell$ entsteht durch die binäre Suche in der Übergangstabelle.

2.8-2 Eigenschaften regulärer Sprachen

„Hat die Sprache E die Eigenschaft P ?“, oder anders ausgedrückt: „Gilt $E \in P$?“ P ist dabei eine Sprachklasse und $E \in \Gamma^*$ die *Beschreibung* einer Sprache $L \subseteq \Sigma^*$ in einem vereinbarten Formalismus, d.h.

$$\Gamma = \Sigma \cup \{ (,), |, \rightarrow, \varepsilon, q_0, S, \dots \} \text{ (z.B. als Typ3-Grammatik oder DFA oder ...)}$$

P heißt **entscheidbar** für reguläre Sprachen, wenn die Existenz eines Algorithmus bewiesen werden kann, der bei Eingabe $E \in \text{REG}(\Sigma)$ die richtige Antwort liefert.

2.8-3 Umwandlung der verschiedenen Beschreibungsformen

Vom DFA zur Grammatik

- Linearzeit: jeden Tabelleneintrag $\delta(q, \sigma) = q'$ als Regel $q \rightarrow \sigma q'$ bzw. $q \rightarrow \sigma$ umschreiben

Von der Grammatik zum NFA

- Linearzeit: jede Regel $q \rightarrow \sigma q'$ als Element von $\Delta(q, \sigma)$ erfassen

Vom regulären Ausdruck zum NFA

- Linearzeit: Shunting-yard-Algorithmus (Auswertung eines Operators bedeutet, Start- und Endzustände normierter Operandenautomaten geeignet "verschalten" \Rightarrow jeweils konstanter Aufwand)

Vom NFA zum DFA (Potenzmengenkonstruktion)

- erfordert mindestens $2^{|Q|}$ Schritte

Leerheitsproblem und Endlichkeitsproblem

2.8-4

Gegeben eine Beschreibung einer regulären Sprache L .

- Gilt $L \neq \emptyset$?
- Gilt $|L| < \infty$?

O.B.d.A sei L durch einen NFA mit ℓ Zuständen gegeben.

Entscheidung des Leerheitsproblem

Entscheide mittels Tiefensuche, ob es im Zustandsdiagramm gerichteten Weg der Länge $\leq \ell$ von einem Startzustand zu einem Endzustand gibt.

Entscheidung des Endlichkeitsproblems

$|L| = \infty$ gdw. ein von einem Startzustand aus erreichbarer Zyklus existiert, von dem aus ein $q \in F$ erreicht werden kann

Teste (jeweils durch Tiefensuche in $O(\ell^2)$ Schritten) die Startzustände darauf, ob ein solcher Zyklus existiert.

mit dem Hase-und-Igel-Algorithmus
(siehe Wikipedia) geht das sogar in

Das Schnittproblem und das Äquivalenzproblem

Zeit $O(\ell)$
2.8-5

Gegeben Beschreibungen regulärer Sprachen $L_1, L_2 \subseteq \Sigma^*$, entscheide

ob $L_1 \cap L_2 = \emptyset$ bzw. ob $L_1 = L_2$ gilt.

Beides kann **reduziert** werden auf das Leerheitsproblem: Berechne Beschreibungen (naheliegender) weiterer regulärer Sprache L, L' mit

$$L_1 \cap L_2 = \emptyset \Leftrightarrow L = \emptyset \text{ bzw. } L_1 = L_2 \Leftrightarrow L' = \emptyset.$$

Reduktion des Schnittproblems auf das Leerheitsproblem

Ausgehend von DFAs die L_1 bzw. L_2 erkennen, konstruiere den *Produktautomaten*, der genau $L := L_1 \cap L_2$ erkennt (siehe Paragraph 2.5-10).

Reduktion des Äquivalenzproblems auf das Leerheitsproblem

$\text{REG}(\Sigma)$ ist abgeschlossen unter Schnitt, Komplement und Vereinigung. Beschreibe entsprechend $L' := (L_1 \setminus L_2) \cup (L_2 \setminus L_1)$.

Alternative Lösung des Äquivalenzproblems

Prüfe $\text{minDFA}(L_1)$ und $\text{minDFA}(L_2)$ auf Isomorphie.

Bemerkung

Das Äquivalenzproblem für reguläre Ausdrücke ist NP-*schwer*, d.h. es stehen nur Exponentialzeitalgorithmen dafür zur Verfügung.

3 Kontextfreie Sprachen und Kellerautomaten

3.1 Chomsky-Typen

Grundlegende Definitionen: siehe Anfang von Abschnitt 2.6

3.1-1 Übersicht über die Chomsky-Grammatiktypen

Definition Jede Chomsky-Grammatik (V, Σ, P, S) ist vom **Typ 0**. Sie ist vom

Typ 1 falls $|\ell| \leq |r|$ für jede Produktion $\ell \rightarrow r$ gilt,

Typ 2 falls zusätzlich $\ell \in V$ für jede Produktion $\ell \rightarrow r$ gilt,

Typ 3 falls zusätzlich $r \in \Sigma \cup \Sigma V$ für jede Produktion $\ell \rightarrow r$ gilt.

Offensichtlich gilt die Implikationskette

$$\text{Typ 3} \Rightarrow \text{Typ 2} \Rightarrow \text{Typ 1} \Rightarrow \text{Typ 0}.$$

Definition (Sprachtypen)

$L \subseteq \Sigma^*$ heißt **Typ i -Sprache**, wenn sie von einer Typ i -Grammatik erzeugt wird. \mathcal{L}_i bezeichnet die entsprechende Sprachklasse. Die **Chomsky-Hierarchie** ist die Kette der Inklusionen

$$\mathcal{L}_3 \subseteq \mathcal{L}_2 \subseteq \mathcal{L}_1 \subseteq \mathcal{L}_0,$$

deren Echtheit wir nach und nach zeigen.

3.1-2 Entscheidbarkeit des Wortproblems

ε kann nicht erzeugt werden durch Typ i -Grammatiken, $i \geq 1$. Deswegen wird die Zusatzregel $S \rightarrow \varepsilon$ erlaubt, falls S auf keiner rechten Regelseite vorkommt. Dadurch bleiben Ableitungen **nichtverkürzend**: Falls $\alpha \Rightarrow \beta$, so $|\alpha| \leq |\beta|$.

Satz

Das Wortproblem für Grammatiken G vom Typ 1 oder größer ist entscheidbar.

Beweis (falls Eingabe $x \neq \varepsilon$): Ein Sättigungsalgorithmus

Systematisch die in 1, 2, 3, ... Schritten aus S ableitbaren Satzformen mit Länge $\leq |x|$ bestimmen und testen, ob x dabei. Stoppe spätestens, wenn keine weiteren Wörter ableitbar. \square

```
def generated_by(x,G):          # x = Eingabewort, G = Typ 1-Grammatik
    n = len(x)
    (A,B) = (set(),set(G['S'])) # A = leere Menge, B = {Startsymbol}
    while A != B:               # Solange es noch Aenderungen gibt, ..
        A = B
        for v in B:             # .. bestimme Satzformen, die aus vorhandenen ..
            for w in derive(v,G): # .. DIREKT ableitbar sind, ergaenze B damit ..
                if len(w) <= n: B.add(w) # .., falls sie nicht zu lang sind.
    return (x in A)             # Teste, ob x dabei ist.
```

ε -Produktionen bei Typ 2 und 3

3.1-3

Bei diesen sind *beliebige* Regeln der Form $A \rightarrow \varepsilon$ erlaubt. Durch Umschreiben kommt man nämlich mit $S \rightarrow \varepsilon$ als *einzigster* ε -Regel aus. Genauer:

Lemma (ε -Elimination)

Jede Typ 2-Grammatik G mit $\varepsilon \notin L(G)$ ist äquivalent zu einer ohne ε -Regeln.

Beweis (Umschreiben zu ε -freier Grammatik)

1. **Nullierbare** Variablen $V_\varepsilon = \{Z \in V : Z \Rightarrow^* \varepsilon\}$ rekursiv ermitteln:

- falls $(Z \rightarrow \varepsilon) \in P$, so gilt $Z \in V_\varepsilon$,
- falls $(Y \rightarrow Z_1 Z_2 \dots Z_k) \in P$ mit $Z_i \in V_\varepsilon$, so $Y \in V_\varepsilon$.

Durch Induktion zeigt man: weitere nullierbare Variablen gibt es nicht.

2. Entferne alle $Z \rightarrow \varepsilon$ aus P .
3. Solange es Änderungen gibt: Zu Regel $B \rightarrow \alpha Z \beta$ mit $Z \in V_\varepsilon, \alpha \beta \in (V \cup \Sigma)^+$ die Regel $B \rightarrow \alpha \beta$ hinzunehmen ("Vorwegnahme der Nullierung").

Beispiel

$S \rightarrow SBc, S \rightarrow BsB, B \rightarrow b, B \rightarrow \varepsilon$ wird ersetzt durch

$S \rightarrow SBc, S \rightarrow Sc, S \rightarrow BsB, S \rightarrow Bs, S \rightarrow sB, B \rightarrow b$.

3.2 Eigenschaften kontextfreier Grammatiken

Typ 2: Kontextfreie Grammatiken (CFG)

3.2-1

Typ 2-Grammatiken heißen auch **kontextfrei**: Die einzige Bedingung ist, dass linke Regelseiten aus einem einzelnen Nonterminal bestehen.

Beispiel (vgl. Paragraph 2.6-4)

Grammatik G_{arith} für arithmetische Ausdrücke über $\Sigma = \{ (,), a, +, \cdot \}$:

$$\begin{array}{lcl} S & \rightarrow & T \\ T & \rightarrow & F \\ F & \rightarrow & a \end{array} \quad \left| \quad \begin{array}{lcl} S & \rightarrow & S + T \\ T & \rightarrow & T \cdot F \\ F & \rightarrow & (S) \end{array}\right.$$

Kontextfreie Sprachen

Von solchen Grammatiken erzeugte Sprachen heißen **kontextfrei**. Die Klasse \mathcal{L}_2 der kontextfreien Sprachen wird auch mit **CFL** bezeichnet. , bei Beschränkung auf Alphabet Σ mit $\text{CFL}(\Sigma)$.

3.2-2 Schreibvereinfachungen und typische Beispiele

Backus-Naur-Form (BNF)

Ersetzungsregel der Gestalt $A \rightarrow \beta_1 | \dots | \beta_n$ steht für $A \rightarrow \beta_1, \dots, A \rightarrow \beta_n$

Übliche Erweiterungen (**EBNF**) und Konventionen sind z.B.

- optionale Bestandteile $[\dots]$,
- mindestens einmal $\{\dots\}^+$, oder beliebig wiederholbare Bestandteile $\{\dots\}^*$,
- $::=$ oder $=$ oder $:$ anstelle \rightarrow .

Einfaches Beispiel

arithmetische Ausdrücke $G_{\text{arith}} : S \rightarrow T | S + T, T \rightarrow F | T \cdot F, F \rightarrow a | (S)$

Beispiel AsciiMath

```
v ::= [A-Za-z] | greek letters | numbers | other constant symbols
u ::= sqrt | text | bb | other unary symbols for font commands
b ::= frac | root | stackrel | other binary symbols
l ::= ( | [ | { | (: | { : | other left brackets
r ::= ) | ] | } | :) | :} | other right brackets
S ::= v | lEr | uS | bSS           Simple expression
I ::= S_S | S^S | S_S~S | S       Intermediate expression
E ::= IE | I/I                     Expression
```

3.2-3 Dyck-Sprachen sind kontextfrei aber nicht regulär

Standardbeispiel

Die nicht-reguläre Sprache $L = \{a^n b^n : n \in \mathbb{N}\}$ wird durch eine CFG erzeugt:

$$S \rightarrow \varepsilon | aSb.$$

Folgerung

BNF und Konventionen
(Großbuchstaben = Nonterminale, S
= Startsymbol) erlauben,
Grammatiken allein durch die
Regelmenge anzugeben.

Echte Inklusion von Chomsky-Klassen: $\mathcal{L}_3 \subset \mathcal{L}_2$.

Alle Dycksprachen sind kontextfrei

Korrekte Klammerungen

siehe Paragraph 2.2-9

- über einem Klammertyp: $S \rightarrow \varepsilon | SS | (S)$
- über zwei Klammertypen: $S \rightarrow \varepsilon | SS | S$

weitere analog

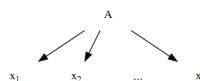
Ableitungsbäume

3.2-4

Für die Definition eines Ableitungsbaums brauchen wir den Begriff des *geordneten Wurzelbaums*. Das ist ein Baum mit ausgezeichnete Wurzel, dessen Kanten von der Wurzel weg gerichtet sind, und dessen von einem Knoten ausgehende Kanten von 1 fortlaufend durchnummeriert sind.

Schließlich ist noch diese Sprechweise hilfreich:

Zu $A \rightarrow x = x_1 \dots x_t$ betrachte den „Regelbaum“ $T_{A \rightarrow x}$: Das ist ein geordneter Wurzelbaum, und hat Tiefe 1, mit A markierte Wurzel und „Blattwort“ x .



Blattwort = fortlaufend gelesene
Markierungen der Blätter

Ableitungsbaum (Syntaxbaum)

Der Ableitung $\text{Abl} : S = \alpha_0 \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n = w \in L(G)$ wird durch Einhängen von Regelbäumen ein Baum (ebenfalls ein geordneter Wurzelbaum) T_{Abl} zugeordnet:

Initialisiere durch Wurzel, die mit S markiert ist

Iteriere über $i = 1, \dots, n$ Wird bei $\alpha_{i-1} \Rightarrow \alpha_i$ die Regel $B \rightarrow \beta$ benutzt, so hänge ein Exemplar von $T_{B \rightarrow \beta}$ an *das* Blatt des aktuellen Baums an, dessen Markierung B gerade ersetzt wird.

Beobachtung

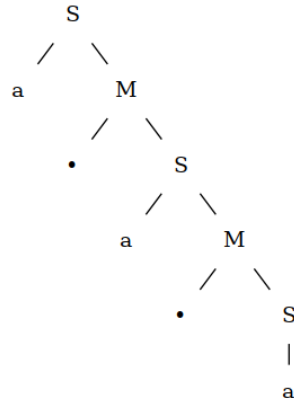
T_{Abl} ist eindeutig durch Abl bestimmt und hat genau $n = |\text{Abl}|$ innere Knoten.

3.2-5 Beispiel 1

Zugehöriger Ableitungsbaum:

In der Grammatik $S \rightarrow a|aM, M \rightarrow \cdot S$ hat $a \cdot a \cdot a$ die einzig mögliche Ableitung

$S \Rightarrow aM \Rightarrow a \cdot S \Rightarrow a \cdot aM \Rightarrow a \cdot a \cdot S \Rightarrow a \cdot a \cdot a$



Beobachtung

Offenbar hat *jedes* von einer regulären Grammatik erzeugte Wort nur eine einzige Ableitung.

3.2-6 Beispiel 2

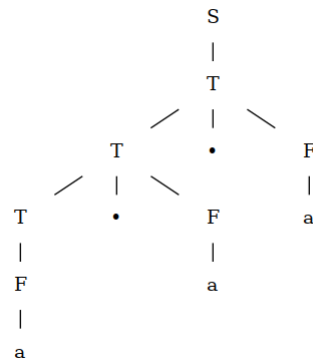
Die Ersetzungsreihenfolge ist *nicht* im Ableitungsbaum gespeichert.

$G_{\text{arith}} : S \rightarrow T|S + T, T \rightarrow F|T \cdot F, F \rightarrow a|(S)$ erlaubt verschiedene Ableitungen mit gleichem Ableitungsbaum:

(L) $S \Rightarrow T \Rightarrow T \cdot F \Rightarrow T \cdot F \cdot F \Rightarrow F \cdot F \cdot F \Rightarrow a \cdot F \cdot F \Rightarrow a \cdot a \cdot F \Rightarrow a \cdot a \cdot a$

und

(R) $S \Rightarrow T \Rightarrow T \cdot F \Rightarrow T \cdot a \Rightarrow T \cdot F \cdot a \Rightarrow T \cdot a \cdot a \Rightarrow F \cdot a \cdot a \Rightarrow a \cdot a \cdot a$



Folgerung

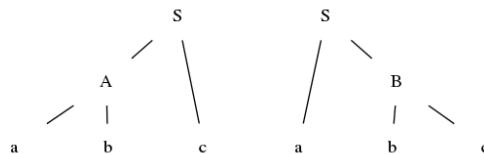
- $w \in L(G) \Leftrightarrow$ es gibt eine Ableitung von w
- \Leftrightarrow es gibt einen Ableitungsbaum mit Blattwort w
- \Leftrightarrow es gibt eine Links- und eine Rechtsableitung von w

3.2-7 Mehrdeutigkeit

Grammatiken, bei denen ein Wort sogar verschiedene *Ableitungsbäume* haben kann, heißen **mehrdeutig**, (offensichtlich ist das nur möglich, wenn diese Wörter verschiedene Linksableitungen besitzen) andere **eindeutig**. Jede reguläre Grammatik ist eindeutig, aber nicht jede kontextfreie:

Beispiel

Die Grammatik $S \rightarrow aB|Ac, A \rightarrow ab, B \rightarrow bc$ ist mehrdeutig:



Eine inhärent mehrdeutige kontextfreie Sprache

Sprachen, die nicht durch eindeutige Grammatiken erzeugt werden *können*, heißen **inhärent mehrdeutig**. Beispiel:

$$L = \{a^i b^j c^k : i = j \text{ oder } j = k\}.$$

Anderer Syntaxbaum \Rightarrow andere Semantik

3.2-8

Dangling-else-Problem: if a < b: if c < d: x = y else: x = z

Frage: Wozu gehört **else**? Antwort:

- Die if-Syntax von Python fordert hier Zeilenwechsel+Einrückung nach der Bedingung, verbietet daher die Konstruktion:

```
if_stmt: 'if' test ':' suite ('elif' test ':' suite)* ['else' ':' suite]
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
```

- Beim Java- (oder C/C++-)Äquivalent würde **else** zum zweiten **if** gehören

Compiler müssen auf eindeutigen Grammatiken beruhen.

Grammatik-Eindeutigkeit ist ein wichtiges Entscheidungsproblem!

Spaß für Linguisten und Lyriker

Beispiel: Ich sah den Mann auf dem Berg mit dem Fernrohr.

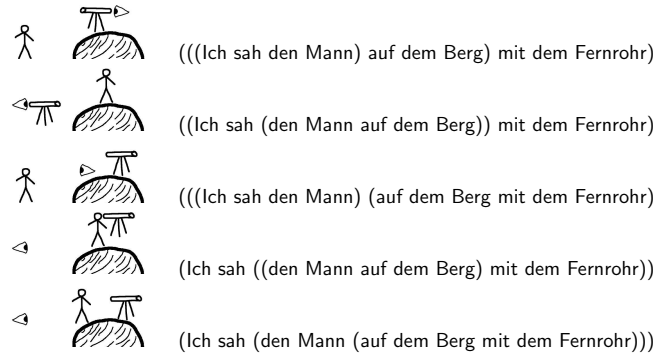
Frage 1 Wer hat das Fernrohr? Ich, der Mann, oder der Berg?

Frage 2 Wieviele verschiedene Deutungen gibt es?

Antwort: 5 = Catalan-Zahl C_3 , allgemein $C_n =$ Anzahl der verschiedenen Binärbäume mit n inneren Knoten $= \frac{1}{n+1} \binom{2n}{n}$

Ich sah den Mann ...

und was soll es bedeuten?



5 Klammerungen

3.3 Chomsky-Normalform und Pumping-Lemma

3.3-1 Chomsky-Normalform

Eine CFG G mit $\varepsilon \notin L(G)$ heie ε -frei. Sie heit Chomsky-Normalform-Grammatik (ChNFG), falls jede Regel eine der beiden Formen hat

$$A \rightarrow BC \text{ oder } A \rightarrow x.$$

Satz

Jede ε -freie CFG G lsst sich umschreiben in eine ChNFG G' mit $L(G) = L(G')$.

Beweis

1. ε -Elimination ausfhren (siehe Paragraph 3.1-3)
2. Kettenregeln $Variable_1 \rightarrow Variable_2$ beseitigen: jede solche Regel entfernen und stattdessen fr jede Regel $Variable_2 \rightarrow \gamma$ die Regel $Variable_1 \rightarrow \gamma$ hinzufgen
3. Terminale separieren: jedes Terminal x in Kombinationen mit anderen Zeichen durch neues Nonterminal V_x ersetzen und Regel $V_x \rightarrow x$ ergnzen
4. „lange Regeln“ $A \rightarrow B_1 B_2 \dots B_n, n > 2$ ersetzen durch $A \rightarrow A_{n-1} B_n, A_{n-1} \rightarrow A_{n-2} B_{n-1}, \dots, A_2 \rightarrow B_1 B_2$. □

3.3-2 Beispielgrammatik: $S \rightarrow AB|ABA, A \rightarrow aA|a, B \rightarrow Bb|b$

ε -Elimination

entferne $B \rightarrow \varepsilon$, nimm Regel vorweg in allen rechten Seiten, die B enthalten

$$S \rightarrow AB|A|ABA|AA \quad A \rightarrow aA|a, B \rightarrow Bb|b$$

Kettenregeln beseitigen

entferne $S \rightarrow A$ und für jede Produktion $A \rightarrow \gamma$ füge $S \rightarrow \gamma$ hinzu.

$$S \rightarrow AB|aA|a|ABA|AA \quad A \rightarrow aA|a, B \rightarrow Bb|b$$

Terminale separieren

ersetze a, b neben Nonterminalen durch V_a, V_b und ergänze $V_a \rightarrow a, V_b \rightarrow b$

$$S \rightarrow AB|V_aA|a|ABA|AA \quad A \rightarrow V_aA|a, B \rightarrow BV_b|b, V_a \rightarrow a, V_b \rightarrow b$$

„Lange Regeln“ ersetzen

ersetze $S \rightarrow ABA$ durch $S \rightarrow S_2A, S_2 \rightarrow AB$

$$S \rightarrow AB|V_aA|a|S_2A|AA, S_2 \rightarrow AB \quad A \rightarrow V_aA|a, B \rightarrow BV_b|b, V_a \rightarrow a, V_b \rightarrow b$$

Eine Beispielableitung in der Originalgrammatik: $S \Rightarrow ABA \Rightarrow aABA \Rightarrow aaBA \Rightarrow aaBbA \Rightarrow aaBbbA \Rightarrow aabbA \Rightarrow aabbaA \Rightarrow aabbaa$

Und ihre Entsprechung in der neuen Grammatik: $S \Rightarrow S_2A \Rightarrow ABA \Rightarrow V_aABA \Rightarrow aABA \Rightarrow aaBA \Rightarrow aaBV_bA \Rightarrow aabV_bA \Rightarrow aabbA \Rightarrow aabbV_aA \Rightarrow aabbaA \Rightarrow aabbaa$

Syntaxbäume von ChNF-Grammatiken

3.3-3

$$\begin{aligned} S &\Rightarrow S_2A \\ &\Rightarrow ABA \\ &\Rightarrow V_aABA \\ &\Rightarrow aABA \\ &\Rightarrow aaBA \\ &\Rightarrow aaBV_bA \end{aligned}$$

$\Rightarrow aabV_bA$

$\Rightarrow aabbA$

$\Rightarrow aabbV_aA$

$\Rightarrow aabbaA$

$\Rightarrow aabbbaa$

$S \Rightarrow S_2A$

$\Rightarrow ABA$

$\Rightarrow V_aABA$

$\Rightarrow aABA$

$\Rightarrow aaBA$

$\Rightarrow aaBV_bA$

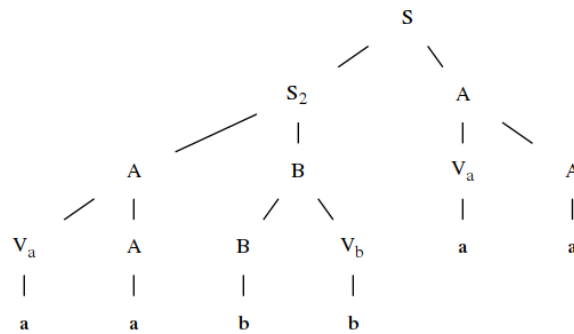
$\Rightarrow aabV_bA$

$\Rightarrow aabbA$

$\Rightarrow aabbV_aA$

$\Rightarrow aabbaA$

$\Rightarrow aabbbaa$



Beobachtungen

Es gibt nur Regeln der Form $A \rightarrow BC$ bzw. $A \rightarrow x$. ChNF-Syntaxbäume sind daher außer im letzten Ableitungsschritt volle Binärbäume.

Das Blattwort eines ChNF-Syntaxbaums der Höhe h hat Länge $\leq 2^{h-1}$.

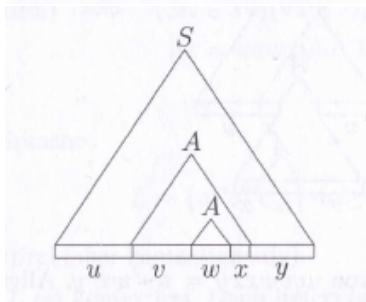
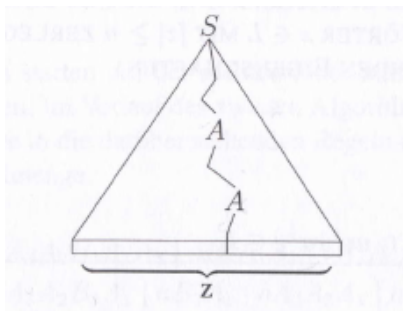
ChNF-Syntaxbäume von Wörtern der Länge $\geq 2^v$ haben Höhe $\geq v + 1$.

3.3-4 Lange Wörter \Rightarrow hohe Syntaxbäume

Sei $G = (V, \Sigma, P, \Sigma)$ ChNF-Grammatik und $z \in L(G)$, $|z| \geq n := 2^{|V|}$. Dann hat

- jeder Syntaxbaum für z Höhe $\geq |V| + 1$ und
- einen Wurzel \rightsquigarrow Blatt-Pfad p mit doppeltem Nonterminal.

(*) Wir betrachten die *letzte* solche Situation entlang p : Nonterminal A kommt doppelt vor, aber dazwischenliegende und nachfolgende Nonterminale nur einfach.

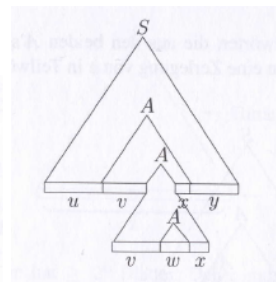
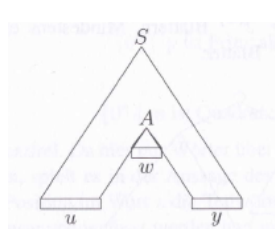
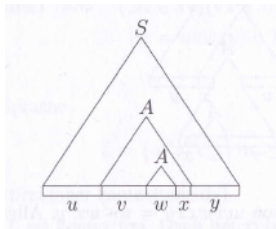


- p benutzt mindestens eine Produktion der Form $A \rightarrow BC$.
- Wegen $\varepsilon \notin L(G)$ können nicht v und x leer sein, d.h. $vx \in \Sigma^+$.
- Wegen (*) ist das obere A maximal $|V| + 1$ Schritte von den Blättern entfernt, also $|vwx| \leq 2^{|V|}$.

Grafiken aus: [Schöningh]

Das Pumping-Lemma für kontextfreie Sprachen

3.3-5



Satz (Pumping-Lemma für kontextfreie Sprachen)

Jedes $L \in CFL(\Sigma)$ erfüllt die „kontextfreie Pumping-Bedingung“ $\text{Cond}_{CFL}(L)$: es gibt eine Zahl $n \geq 1$, so dass jedes Wort $z \in L$ der Länge $\geq n$ zerlegbar ist als

$$z = uvwxy \text{ für geeignete } u, v, w, x, y \in \Sigma^* \text{ mit } |vx| \geq 1 \text{ und } |vwx| \leq n,$$

und für diese Zerlegung gilt $uv^iwx^iy \in L$ für alle $i \geq 0$. \square

Negierte Pumpbedingung und Prover-Verifier-Game

3.3-6

$$\neg \text{Cond}_{CFL}(L) \iff \forall n \exists z \in L \wedge |z| \geq n \forall u, v, w, x, y : z = uvwxy \wedge |vwx| \leq n \wedge |vx| \geq 1 \wedge \exists i \geq 0 : uv^iwx^iy \notin L \Rightarrow L \notin \mathcal{L}_2$$

Vergleich: Schach

Schwarz hat Gewinnstrategie beim Schach, falls

∀ Züge von Weiß ∃ Zug von Schwarz ∀ Züge von Weiß ∃ Zug von Schwarz ..., so dass der weiße König im Schach steht.

Falls $\exists \forall \exists \forall \dots$ schwarzer König im Schach ... hat Weiß eine Gewinnstrategie.

Beispiel 1: $L_{abc} = \{a^n b^n c^n | n \in \mathbb{N}\}$ ist nicht kontextfrei.

Beweis

- sei n eine beliebige Zahl
- betrachte $z = a^n b^n c^n \in L$
- sei $z = uvwxy$ eine beliebige Zerlegung mit $|vx| \geq 1$ und $|vwx| \leq n$
- betrachte $i = 0$ und $uv^0wx^0y = uwy$: wegen $|vx| \leq |vwx| \leq n$ enthält vx nicht a und c , also $uwy \notin L$. \square

Beispiel 1: $L_{abc} = \{a^n b^n c^n | n \in \mathbb{N}\}$ ist nicht kontextfrei.

Beweis durch Entlarven eines betrügerischen Provers

Prover nennt eine (angebliche) Pumping-Zahl n .

Verifier wählt das Wort $z = a^n b^n c^n \in L$. (der entscheidende Schritt!)

Prover nennt eine Zerlegung $z = uvwxy$ mit $|vx| \geq 1$ und $|vwx| \leq n$.

Verifier wählt $i = 0$ und $uv^0wx^0y = uwy$: wegen $|vx| \leq |vwx| \leq n$ enthält vx nicht a und c , also $uwy \notin L$. \square

3.3-7 Beispiel 2: „geankerte“ Palindrome (Prover gewinnt)

Sei $L = \{w\%w^R | w \in \{0,1\}^*\}$.

Bestehen jeder Probe

Prover nennt $n = 3$.

Verifier wählt irgendein $z \in L, |z| \geq n$.
(Offenbar: $z = ta\%at^R$ für ein $t \in \{0,1\}^*$ und ein $a \in \Sigma$.)

Prover nennt die Zerlegung $z = uvwxy$ mit
 $u = t, v = a, w = \%, x = a, y = t^R$. Es gilt $|vwx| \leq 3, |vx| > 0$

Verifier wählt irgendein $i \geq 0$ und findet keinen Widerspruch:
 $uv^iwx^iy \in L$

Prover gewinnt so immer, denn L ist kontextfrei: $S \rightarrow \%0S0|1S1$.

Bemerkung

Die reguläre Pumpbedingung $\text{Cond}_{\text{REG}}(L)$ (siehe Paragraph 2.2-6) hat gleiche Quantoren-Struktur und es gelten gleiche Zusammenhänge zwischen P.-V.-Gewinnstrategien und Erfüllung der Bedingung.

Zur Echtheit der Chomsky-Hierarchie

Es gilt $\mathcal{L}_2 \subset \mathcal{L}_1$.

$L_{abc} = \{a^n b^n c^n | n \in \mathbb{N}\}$ ist nicht kontextfrei, aber es gibt eine Typ 1-Grammatik, die L_{abc} generiert. Sehen wir uns später an ...

Notwendig aber nicht hinreichend!

Kontextfrei \Rightarrow „pumpbar“, aber die Umkehrung gilt nicht, dafür gibt es Gegenbeispiele!

3.4 Der CYK-Algorithmus

Das Wortproblem für kontextfreie Sprachen

3.4-1

gegeben $G = (V, \Sigma, P, S)$ ChNF-Grammatik und $x \in \Sigma^*$

gefragt Gilt $x \in L(G)$? bzw. äquivalent: Gilt $S \Rightarrow^* x$?

Rekursiver Entscheidungsalgorithmus

Basisfall $|x| = 1$, d.h. $x = x_1 \in \Sigma$.

Dann $A \Rightarrow^* x \Leftrightarrow A \rightarrow x_1 \in P$

Rekursionsfall $|x| \geq 2$, d.h. $x = x_1 x_2 \dots x_n, x_i \in \Sigma$.

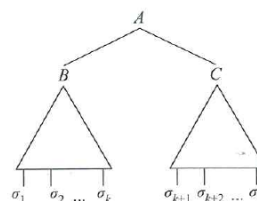
Dann $A \Rightarrow^* x \Leftrightarrow \exists$ Produktion $A \rightarrow BC \in P$

und

$\exists k \in \{1, 2, \dots, n\}$ so dass

(1) $B \Rightarrow^* x_1 \dots x_{k-1}$ und

(2) $C \Rightarrow^* x_k \dots x_n$

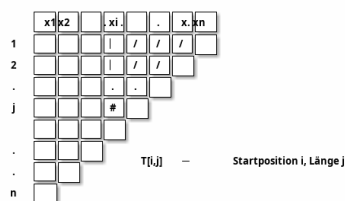


Memoisation

3.4-2

- Wir brauchen immer wieder die Information, aus welchen Nonterminalen *Teilwörter* von x generiert werden können.
- Die Aufrufergebnisse werden zwischengespeichert in einer Tabelle $T[i, j] := \{A \in V : A \Rightarrow^* \underbrace{x_i x_{i+1} \dots x_{i+j-1}}_{\text{Länge } j}\}$.

Für $i + j > n + 1$ ist $T[i, j] = \emptyset$, d.h. T ist "dreieckig:



3.4-3 Füllen der Tabelle

Basisfall

$$A \rightarrow x_i \in P \Leftrightarrow \text{Nonterminal } A \in T[i, 1]$$

0	x1	x2	.	xi	.			xn
1								

Rekursionsfall

$$\begin{aligned}
A \Rightarrow^* x_i \dots x_{i+j-1} &\Leftrightarrow \exists \text{Regel } A \rightarrow BC \in P \\
\exists k : B &\Rightarrow^* x_i \dots x_{i+k-1} \wedge C \Rightarrow x_{i+k} \dots x_{i+j-1} \\
\text{das hei\u00dft:} \\
A \in \textcolor{red}{T}[i, j] &\Leftrightarrow \exists A \rightarrow BC \in P \\
\exists k : B \in \textcolor{teal}{T}[i, k] \wedge C &\in \textcolor{blue}{T}[i+k, j-k]
\end{aligned}$$

Figure 1 shows a 10x10 grid representing a sparse matrix. The columns are labeled at the top: i , $i+h1+k$, $i+j$, and $x1$ through xn . The rows are labeled on the left: 0 , k , and j . The grid shows a pattern of colored cells: a red cell at (j, i) , a green cell at $(k, i+h1+k)$, and a blue cell at $(k, i+j)$. Other cells are white or light gray.

Entscheidung

$S \Rightarrow^* x \Leftrightarrow S \in T[1, n]$
d.h. $T[1, n]$ ist der „kritische Eintrag“ der Tabelle.

3.4-4 Der CYK-Algorithmus (Cocke-Younger-Kasami)

1. erzeuge CYK-Tabelle T zu ChNFG und Eingabe x und fülle sie
2. liefere **True** zurück, falls die kritische Zelle das Startsymbol enthält.

Python-Pseudocode

```

1 for i in range(n):
2     t[i,1] = {"Nonterminale A mit: A->x[i] in P"} # Basisfall
3 for j in range(2,n+1): # Rekursion: Teilwoerter der Laengen j = 2..n
4     for i in range(2,n-j+1): # moegl. Anfangspositionen i: 0, 1, ..., n-j
5         t[i,j] = set({}) # leere Menge
6         for k in range(1,j): # moegliche Teillaengen k: 1, 2, ..., j-1
7             t[i,j].update({"A mit: A->BC in P und B in t[i,k] und C in t[i+k,j-k]"}))

```

Für bessere Lesbarkeit steht in Zeilen 2 und 7 nur „Pseudopython“, funktionierender Code steht in der Datei `Def_Grammars.py` im `ti-notebooks`-Repository.

Laufzeit

Es gibt drei verschachtelte **for**-Schleifen mit Länge $\leq n$. Die Laufzeit ist daher $O(n^3)$ — die Größe der Regelmenge ist im „Groß-Oh“ enthalten.

3.4-5 Beispiel

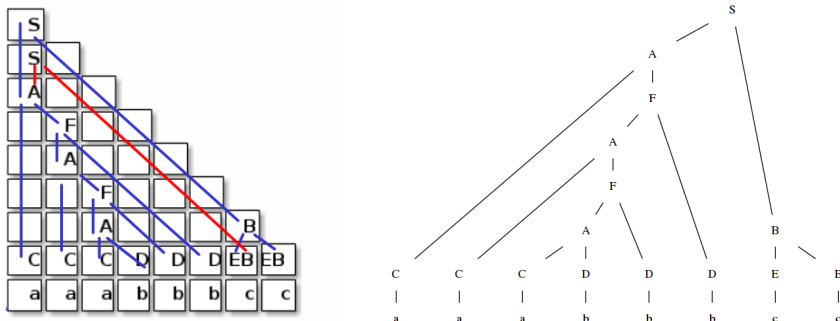
- Ausgangsregeln $S \rightarrow AB, A \rightarrow ab|aAb, B \rightarrow c|xB$, generiert mit Startsymbol S die Sprache $L = \{a^n b^n c^m | n, m \geq 1\}$

- | | | | | | |
|---|---|----|---|---|----|
| S | → | AB | C | → | a |
| A | → | CD | D | → | b |
| A | → | CF | E | → | c |
| B | → | c | F | → | AD |
| B | → | EB | | | |

-
- The figure consists of two 8x8 grids. The left grid shows a staircase pattern of item labels: Row 1: a, a, a, b, b, c, c, c; Row 2: C, C, C, D, D, E, E, E; Row 3: F, F, F, F, F, F, F, F; Row 4: A, A, A, A, A, A, A, A; Row 5: S, S, S, S, S, S, S, S; Row 6: S, S, S, S, S, S, S, S; Row 7: S, S, S, S, S, S, S, S; Row 8: S, S, S, S, S, S, S, S. The right grid shows a similar staircase pattern: Row 1: a, a, a, b, b, b, c, c; Row 2: C, C, C, D, D, D, E, E; Row 3: F, F, F, F, F, F, F, F; Row 4: A, A, A, A, A, A, A, A; Row 5: S, S, S, S, S, S, S, S; Row 6: S, S, S, S, S, S, S, S; Row 7: S, S, S, S, S, S, S, S; Row 8: S, S, S, S, S, S, S, S.

3.4-6

- | | | | | | |
|---|---|----|---|---|----|
| S | → | AB | C | → | a |
| A | → | CD | D | → | b |
| A | → | CF | E | → | c |
| B | → | c | F | → | AD |
| B | → | EB | | | |



3.5 Abschlusseigenschaften

3.5-1

Satz

Die Klasse \mathcal{L}_2 der kontextfreien Sprachen ist (1) abgeschlossen unter *Vereinigung*, *Produkt*, *Stern* aber (2) nicht abgeschlossen unter *Schnitt* und *Komplement*.

Beweis

(1) Seien $G_i = (V_i, \Sigma, P_i, S_i), i = 1, 2$ kontextfreie Grammatiken und $L_i = L(G_i)$. O.B.d.A. sei $V_1 \cap V_2 = \emptyset$ und sei $S \notin V_1 \cup V_2$ ein neues Symbol.

$L_1 \cup L_2, L_1 \cdot L_2$ und L_1^* werden dann offensichtlich generiert durch:

- $G_{\cup} = (V_1 \cup V_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \rightarrow S_1 | S_2\}, S)$
- $G_{\cdot} = (V_1 \cup V_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \rightarrow S_1 S_2\}, S)$
- $G_{*} = (V_1 \cup \{S\}, \Sigma, P_1 \cup \{S \rightarrow \varepsilon | S_1, S_1 \rightarrow S_1 S_1\}, S)$

(2) Der Schnitt der kontextfreien Sprachen $L_1 = \{a^m b^m c^n : m, n \geq 0\}$ und $L_2 = \{a^m b^n c^n : m, n \geq 0\}$ ist bekanntlich nicht kontextfrei.

Wegen $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$ können die kontextfreien Sprachen auch nicht abgeschlossen sein unter Komplement. □

3.5-2 Weitere Abschlusseigenschaften

Satz

\mathcal{L}_2 ist abgeschlossen unter homomorphem Bild und Urbild und unter Reflexion, aber nicht unter Shuffle oder Quotient.

Beweis (nur Abschluss unter homomorphem Bild)

Sei $L \subseteq \Sigma^*$ kontextfrei erzeugt durch (V, Σ, P, S) und $\hat{h} : \Sigma^* \rightarrow \Gamma^*$ homomorphe Fortsetzung von $h : \Sigma \rightarrow \Gamma^*$. Dann erzeugt folgende Typ 2-Grammatik gerade das Bild $\hat{h}(L)$:

$G_h = (V \cup \{V_a : a \in \Sigma\}, \Sigma, P' \cup \{V_a \rightarrow h(a) : a \in \Sigma\})$, wobei P' aus P entsteht, indem man jedes vorkommende Terminal a durch V_a ersetzt. □

Satz (hier ohne Beweis)

\mathcal{L}_2 ist abgeschlossen unter Schnitt und Shuffle mit regulären Sprachen.

3.5-3 Die Struktur kontextfreier Sprachen

Erinnerung (Dycksprachen)

siehe Paragraph 2.2-9

Für $\Sigma_k = \{a_1, b_1, \dots, a_k, b_k\}$ bezeichnet $D_k \subseteq \Sigma_k^*$ die Menge der korrekten Klammerungen mit k Klammertypen.

Folgende Aussage kann als kontextfreie Entsprechung des Satzes von Kleene (Paragraph 2.5-7) angesehen werden:

Satz von Chomsky-Schützenberger (1963)

Zu jeder kontextfreien Sprache $L \subseteq \Sigma^*$ gibt es ein k , eine reguläre Sprache $R \subseteq \Sigma_k^*$ und einen Homomorphismus $h : \Sigma_k^* \rightarrow \Sigma^*$, so dass

$$L = h(R \cap D_k).$$

Anwendungsbeispiel

3.5-4

Behauptung

$L = \{a^m b^n c^{m+n} : m, n > 0\}$ ist kontextfrei.

Beweis

Es genügt, L in der Form $L = h(R \cap D_k)$ darzustellen.

- Umschreibung $a^m b^n c^{m+n} = a^m b^n c^n c^m$
- $R = L(a_1 * b_1 * b_2 * a_2^*)$
- Dycksprache $D_2 \subseteq \{a_1, a_2, b_1, b_2\}^*$
- Schnitt: $R \cap D_2 = \{a_1^n b_1^m b_2^m a_2^n : m, n \geq 0\}$
- Homomorphismus: $a_1 \mapsto a, b_1 \mapsto b, a_2 \mapsto c, b_2 \mapsto c$
- homomorphes Bild des Schnitts: $\{a^m b^n c^n c^m : m, n > 0\}$ □

Beobachtungen zu Pumping-Lemmas für unäre Sprachen

3.5-5

Bei einelementigem Alphabet $\Sigma = \{a\}$ kommt es in den Pumping-Lemmas nicht darauf an, *wo* Pumpfaktoren stehen, nur *wie lang* sie sind. Somit sind in diesem Fall die Aussagen beider Pumping-Lemmas gleich:

Für kontextfreie Sprachen $L \subseteq \{a\}^*$ gilt die „unäre Pumpbedingung“

$$\exists n \forall m : m \geq n \wedge a^m \in L \Rightarrow \exists k, s : m = k + s, 1 \leq s \leq n \wedge \underbrace{a^k (a^s)^\ell}_{L(a^k (a^s)^*) \subseteq L} \in L.$$

Und: $L \subseteq \{a\}^*$ ist *genau dann* regulär, wenn die unäre Pumpbedingung erfüllt ist (Beweis folgt)

Folgerung Jede unäre kontextfreie Sprache ist sogar regulär.

Beispiel: $L = \{a^p : p \text{ ist Quadratzahl}\}$ ist nicht kontextfrei.

Beweis: Für beliebiges n gilt $a^{n^2} \in L$. Aber für beliebige $k, s, 1 \leq s \leq n$ gilt $n^2 < n^2 + s \leq n^2 + n < n^2 + 2s + 1$, d.h. durch „Aufpumpen“ entstünde eine Wortlänge, die keine Quadratzahl ist. □

Unär + Pumpbedingung \Rightarrow regulär — der Beweis

3.5-6

Gelte $\exists n \forall m : m \geq n \wedge a^m \in L \Rightarrow \exists k, s : m = k + s, 1 \leq s \leq n \wedge L(a^k (a^s)^*) \subseteq L$.

Für jedes $m \geq n$ gibt es solche k, s , und wegen $s \leq n$ kommen dabei nur endlich viele Werte s vor: s_1, \dots, s_t . Sei $p = s_1 \cdots s_t$.

siehe auch die Beobachtung in Paragraph 2.2-4

Zu $p' \geq p$ definieren wir die reguläre Sprache $L' \subseteq L$:

$$L' = \{x \in L : |x| < p\} \cup \bigcup_{r: p \leq r \leq p', a^r \in L} L(a^r(a^p)^*).$$

Wegen der Wahl von p gilt $L' \subseteq L$.

Behauptung Falls p' genügend groß ist, so gilt $x \in L \Rightarrow x \in L'$ und damit $L = L'$.

Für x mit $|x| < p$ stimmt das offenbar. Sei $z = a^m \in L$ mit $m \geq p$.

Nach Definition gilt $z \in L'$, falls $\exists a^r \in L, p \leq r \leq p', r \equiv m \pmod p$. Also: wähle p' so, dass Längen p, \dots, p' alle möglichen Reste $\pmod p$ bilden, die bei langen Wörtern aus L überhaupt auftreten. Davon gibt es aber nur endlich viele, also gibt es solches p' . \square

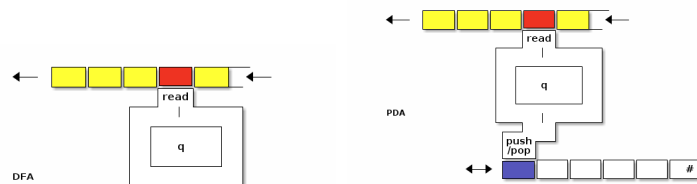
3.6 Kellerautomaten (Pushdown-Automaten)

3.6-1 Motivation

Beispiel: Überprüfen korrekter Klammerungen über $[\cdot]$

$$D_1 = \{x : \text{für jedes Präfix } y \text{ von } x \text{ gilt } \#_[]y \geq \#_[]y \text{ und } \#_[]x = \#_[]x\}$$

DFAs können die Syntax nicht prüfen: sie „können nicht beliebig weit zählen“.



Speicheranforderungen für geeignetes Akzeptormodell (PDA)

- Stack-Zugriff (push/pop) reicht für diese Sprache aus

3.6-2 Definition

Ein nichtdeterministischer **Kellerautomat** (Pushdown-Automat - **PDA**) $P = (Q, \Sigma, \Gamma, \Delta, q_0, z_0, F)$ ist gegeben durch

- endliche Zustandsmenge Q , Startzustand $q_0 \in Q$, Alphabet Σ ,
- *Kelleralphabet* Γ (enthält Σ),
- *Übergangsfunktion* $\Delta : Q \times \Sigma_\epsilon \times \Gamma \rightarrow 2^{Q \times \Gamma^*}$

Erinnerung: Für eine Menge M bezeichnet 2^M deren Potenzmenge

- *Kellerstartsymbol* z_0 und

- Endzustandsmenge F .

Ist in Beispielen nicht anderes angegeben, steht $\#$ für das Kellerstartsymbol.

Aus der Signatur von Δ ergibt sich:

Ausgehend von aktuellem Zustand q , Eingabesymbol σ und Kellersymbol κ ist eine (eventuell leere) Menge von Aktionen festgelegt. Eine Aktion besteht aus

- Zustand aktualisieren $q \mapsto q'$,
- Lesekopf positionieren und
- „Kellerwort“ aktualisieren: *Top-Symbol* κ durch Zeichenfolge ω ersetzen.

Beispiel: Ein PDA für D_1

3.6-3

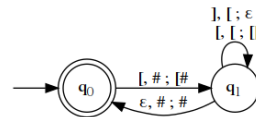
$P_{\text{Dyck}} = (\{q_0, q_1\}, \{[,]\}, \Sigma \cup \{\#\}, \Delta, q_0, \#, \{q_0\})$ mit Übergangstabelle:

$\Delta : (q, \sigma, \kappa) \mapsto \{(q', \omega), \dots\}$	Erklärung
$(q_0, [, \#) \mapsto \{(q_1, [\#])\}$	öffnende Klammer speichern
$(q_1, [,]) \mapsto \{(q_1, [])\}$	öffnende Klammer <i>voranstellen</i> („auftürmen“)
$(q_1,],]) \mapsto \{(q_1, \varepsilon)\}$	schließende Klammer: Turm abbauen
$(q_1, \varepsilon, \#) \mapsto \{(q_0, \#)\}$	bis hierhin ausbalanciert

An allen anderen Stellen liefert Δ die leere Menge.

Maschinenlesbare Darstellung (als Python-Dictionary)

```
P_Dyck = {'Q': {'q1', 'q0'},
'Sigma': {'[', ']'},
'Gamma': {'#', '[', ']'},
'Delta': {('q0', '[', '#'): {'q1', '[#']},
('q1', '[', ']'): {'q1', '[ ]'},
('q1', ']', ']' ): {'q1', ''},
('q1', ']', '#'): {'q0', '#'}},
'q0': 'q0',
'z0': '#',
'F': {'q0'}}
```



Wirkung eines Übergangs

3.6-4

$$\Delta(q, \sigma, \kappa) \ni (q', \omega), \text{ wobei } \omega = \omega_1 \dots \omega_\ell$$

Bedingung	Bedeutung
$\ell = 0$	Top-Symbol löschen (Pop-Operation)
$\ell = 1$	Top-Symbol durch ω_1 ersetzen
$\ell > 1, \omega_\ell = \kappa$	Push von $\omega_1, \dots, \omega_{\ell-1}$
$\ell > 1, \omega_\ell \neq \kappa$	Pop, dann Push von $\omega_1, \dots, \omega_\ell$

Nichtdeterminismus beachten!

- Nichtdeterminismus durch $|\Delta(q, \sigma, \kappa)| > 1$ bei $\sigma \in \Sigma_\varepsilon$
- Nichtdeterminismus durch $|\Delta(q, \sigma, \kappa) \cup \Delta(q, \varepsilon, \kappa)| > 1$ bei $\sigma \neq \varepsilon$
- „Steckenbleiben“ bei $\Delta(q, \sigma, \kappa) \cup \Delta(q, \varepsilon, \kappa) = \emptyset$ für alle $\sigma \in \Sigma$

3.6-5 Konfigurationen

Eine **Konfiguration** (engl. instantaneous description– ID) des PDA P ist ein Tripel $k = (q, v, w) \in Q \times \Sigma^* \times \Gamma^*$ bestehend aus

- *aktuellem Zustand* q ,
- *Restwort* v und
- *Kellerwort* w .

Die **Übergangsrelation** definiert mögliche *Nachfolgerkonfigurationen* k' (Notation $k \vdash k'$):

$$(q, av, sw) \vdash_P (q', v, w'w) \Leftrightarrow (q', w') \in \Delta(q, a, s)$$

und

$$(q, av, sw) \vdash_P (q', av, w'w) \Leftrightarrow (q', w') \in \Delta(q, \varepsilon, s).$$

Grundsätzliche Idee der Akzeptierung

Die Startkonfiguration auf der Eingabe x ist $k_0(x) = (q_0, x, z_0)$.

x wird akzeptiert, wenn eine Kette von Übergängen $k_0(x) \vdash_P k_1 \vdash_P \dots \vdash_P k_F$ bis zu irgendeiner *Endkonfiguration* k_F besteht.

Anders ausgedrückt geht es darum, ob es eine Endkonfiguration k_F gibt, so dass $k_0(x) \vdash_P^* k_F$ gilt (reflexiv-transitiver Abschluss).

3.6-6 Die erkannte Sprache

$$L(P) = \{x \in \Sigma^* : P \text{ akzeptiert } x\}$$

Es gibt zwei formal unterschiedliche Akzeptierungsdefinitionen:

Akzeptieren mit leerem Keller

P **akzeptiert x mit leerem Keller**, falls für ein $q \in Q$ gilt $(q_0, x, z_0) \vdash_P^* (q, \varepsilon, \varepsilon)$.

Akzeptieren mit Endzustand

P **akzeptiert x mit Endzustand**, falls für ein $q \in F$ und $w \in \Gamma^*$ gilt $(q_0, x, z_0) \vdash_P^* (q, \varepsilon, w)$.

3.6-7 Äquivalenz der beiden Akzeptierungsvarianten

Vom „Endzustandsakzeptierer“ zum „Leerkeller-Akzeptierer“

Ergänze Übergänge, die Konfigurationen (q', ε, w) mit $q' \in F$ überführen in $(q'', \varepsilon, \varepsilon)$ („Keller leeren“).

Vom „Leerkeller-Akzeptierer“ zum „Endzustandsakzeptierer“

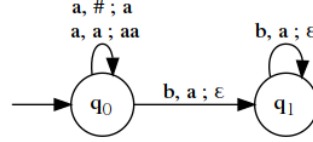
Verwende ein neues Keller-Symbol \perp und zusätzlichen Übergang $\Delta(q', \varepsilon, \perp) = (q_F, \perp)$ mit neuem Zustand q_F und $F = \{q_F\}$.

Beispiel: Leerkellerakzeptierer für $L_{ab} = \{a^n b^n : n > 0\}$

3.6-8

$P_{ab} = (\{q_0, q_1\}, \{a, b\}, \{a, b, \#\}, \Delta, q_0, \#, \emptyset)$ mit Übergangstabelle:

(q, σ, κ)	$\Delta(q, \sigma, \kappa)$
$(q_0, a, \#)$	$\{(q_0, a)\}$
(q_0, a, a)	$\{(q_0, aa)\}$
(q_0, b, a)	$\{(q_1, \varepsilon)\}$
(q_1, b, a)	$\{(q_1, \varepsilon)\}$



An allen anderen Stellen liefert Δ die leere Menge.

Korrektheit der Konstruktion

$a^n b^n$ wird akzeptiert: $(q_0, a^n b^n, \#) \vdash^* (q_0, b^n, a^n) \vdash (q_1, b^{n-1}, a^{n-1}) \vdash^* (q_1, \varepsilon, \varepsilon)$,

aber andere Wörter nicht:

- Übergangsfunktion ist undefiniert, falls b zuerst oder vor a kommt
- bei Eingabe $a^n b^m$ mit $n < m$ ist Keller leer bevor Eingabe abgearbeitet,
- bei Eingabe $a^n b^m$ mit $n > m$ ist Eingabe abgearbeitet bevor Keller leer.

P_{ab} ist (ebenso wie P_{Dyck}) *im wesentlichen* deterministisch: eine einzige Konfigurationsfolge $k_0(x) \vdash k_1 \vdash k_2 \vdash \dots$

Deterministische Kellerautomaten (DPDA)

3.6-9

Definition

Ein PDA $P = (Q, \Sigma, \Gamma, \Delta, q_0, z_0, F)$ heißt **deterministisch**, falls für alle $(q, \sigma, \gamma) \in Q \times \Sigma \times \Gamma$ gilt

$$|\Delta(q, \sigma, \gamma)| + |\Delta(q, \varepsilon, \gamma)| \leq 1.$$

Als Akzeptierungsbegriff ist *Endzustandsakzeptierung* festgelegt.

Bei DPDAs ist es bequem, anstelle des mengenwertigen Δ die *partielle* Überföhrungsfunktion

$$\delta : Q \times (\Sigma_\varepsilon) \times \Gamma \rightrightarrows Q \times \Gamma^*,$$

zu betrachten, mit $\Delta(q, \sigma, \kappa) = \{\delta(q, \sigma, \kappa)\}$ (sofern linke Seite nichtleer).

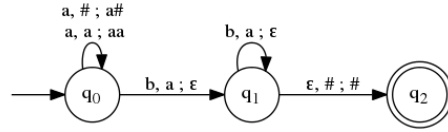
Ein DPDA stoppt, falls für die aktuelle Konfiguration keine Nachfolgekonfiguration definiert ist.

Beispiel: DPDA für L_{ab}

3.6-10

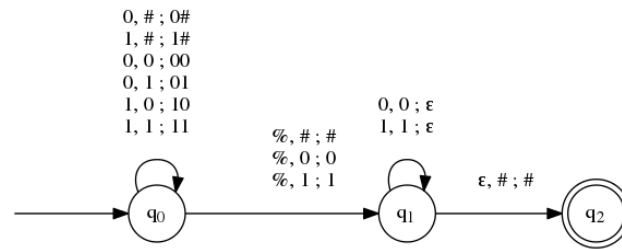
$DP_{ab} = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b, \#\}, \delta, q_0, \#, \{q_2\})$ mit partieller

	(q, σ, κ)	(q', ω)
	$(q_0, a, \#)$	$(q_0, a\#)$
	(q_0, a, a)	(q_0, aa)
Übergangsfunktion δ :	(q_0, b, a)	(q_1, ε)
	(q_1, b, a)	(q_1, ε)
	$(q_1, \varepsilon, \#)$	$(q_2, \#)$



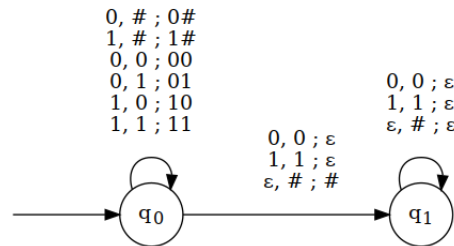
Die anderen Stellen von δ bleiben undefiniert.

3.6-11 **Beispiel: DPDA für „Anker-Palindrome“** $L_{aPal} = \{w\%w^R : w \in \{0,1\}^*\}$
 $DP_{aPal} = (\{q_0, q_1, q_2\}, \{0, 1, \%\}, \{0, 1, \%\# \}, \delta, q_0, \#, \{q_2\})$



Leerkellerakzeptierer für Palindrome gerader Länge

3.6-12 $L_{gPal} = \{ww^R : w \in \{0,1\}^*\}$
 $P_{gPal} = (\{q_0, q_1\}, \{0, 1\}, \{0, 1, \#\}, \Delta, q_0, \#, \emptyset)$



Wegen $\Delta(q_0, \sigma, \sigma) = \{(q_0, \sigma\sigma), (q_1, \varepsilon)\}$ für $\sigma \in \Sigma$ kann der PDA die Mitte von Eingabewörtern $x \in L_{gPal}$ „erraten“ und den Keller abbauen.

Bei $x \notin L_{gPal}$ kann der Keller nicht abgebaut werden.

3.6-13 **Äquivalenzsatz für kontextfreie Sprachen**

Satz

Eine Sprache L ist genau dann kontextfrei, wenn sie von einem PDA erkannt wird.

Beweis: Teil 1 — Typ2-Grammatik $= (V, \Sigma, P, S) \mapsto$
Leerkellerakzeptierer P_G

Betrachte $PDA_G = (\{q_0\}, \Sigma, \Gamma = \Sigma \cup V, \Delta, q_0, z_0 = S)$ mit

- $\forall A \in V : \Delta(q_0, \varepsilon, A) := \{(q_0, w) : A \rightarrow w \in P\}$ und
- $\forall a \in \Sigma : \Delta(q_0, a, a) := \{(q_0, \varepsilon)\}$.

In jedem Schritt agiert der PDA anhand des Top-Symbols κ :

- $\kappa = A \in V$: ersetze A durch rechte Seite einer Produktion $A \rightarrow w \in P$, (spontan, d.h. ohne ein Eingabezeichen zu verarbeiten)
- $\kappa = a \in \Sigma$: nur falls a mit nächstem Eingabezeichen übereinstimmt, kann es gelöscht werden.

Offenbar ist $(q_0, \varepsilon, \varepsilon)$ genau dann erreichbar, wenn der PDA eine Linksableitung des Eingabeworts erraten kann und umgekehrt.

Beispiel: $S \rightarrow A|B, A \rightarrow aAa|aa, B \rightarrow bBb|bb, L = \{a^{2n}, b^{2n} : n \geq 0\}$

Eine mögliche **Berechnung** (Konfigurationsfolge) auf Eingabe aab :

$(q_0, aab, S) \vdash (q_0, aab, A) \vdash (q_0, aab, aAa) \vdash (q_0, ab, Aa) \vdash (q_0, ab, aaa) \vdash (q_0, b, aa)$

Äquivalenzsatz für kontextfreie Sprachen (Forts.)

3.6-14

Teil 2 — PDA \mapsto Typ2-Grammatik (Idee)

Simuliere PDA-Berechnung durch Linksableitungsschritte einer geeigneten Grammatik. □

Folgerung

Jeder PDA kann durch einen PDA mit nur einem Zustand simuliert werden.

Beweis Gegeben PDA P , betrachte äquivalente Typ2-Grammatik G_P und konstruiere zu dieser den äquivalenten PDA_{G_P} wie im Beweis Teil 1.

Deterministisch kontextfreie Sprachen

3.6-15

Definition

Eine formale Sprache L heißt **deterministisch kontextfrei** (Sprachklasse DCFL), wenn es einen DPDA gibt, der sie erkennt.

Beispiele

- L_{ab} und L_{aPal} sind deterministisch-kontextfrei, aber nicht regulär
- L_{gPal} ist kontextfrei, aber nicht *deterministisch-kontextfrei* (ohne Beweis)

Folgerung

$$\text{REG} \subset \text{DCFL} \subset \text{CFL}.$$

Einige Fakten zu DCFL

3.6-16

Abschlusseigenschaften

	abgeschlossen	Beweis
(1) Komplement	ja	technisch recht aufwändig (siehe hier)
(2) Schnitt	nein	gleiches Gegenbsp. wie bei CFL (sh. Par. 3.3-2)
(3) Vereinigung	nein	folgt mit de Morganscher Regel aus (1)+(2)

DCFL sind auch abgeschlossen unter Schnitt mit regulären Sprachen (Beweis mit Produktkonstruktion ähnlich wie in Paragraph 2.5-10 im Skript).

Das Wortproblem ist in Linearzeit entscheidbar

Durch Schritt-für-Schritt-Simulation: Jeder Schritt beinhaltet höchstens konstant viele ε -Übergänge, deswegen ist die Laufzeit linear beschränkt.

Anwendung im Compilerbau: Ableitungen für $x \in L(G)$ in Linearzeit finden.

Genauer wird **reduziert**: *Rückersetzung* bis zum Startsymbol. Das ist in Linearzeit möglich: jede DCFL-Sprache wird erzeugt durch eine **LR(1)-Grammatik** — k.f. Grammatik mit der Eigenschaft, dass beim Lesen von **L**-inks nach rechts **R**-rechtsreduktionen eindeutig bestimmt sind durch **1** Symbol Vorausschau.

3.7 Entscheidungsprobleme für kontextfreie Grammatiken

3.7-1 Entscheidbare Probleme

Das Wortproblem „gegeben (G, w) , gilt $w \in L(G)$?“ ist entscheidbar

z.B. mit dem CYK-Algorithmus.

Das Leerheitsproblem „gegeben G , gilt $L(G) = \emptyset$?“ ist entscheidbar:

Man bestimmt zunächst die **terminierbaren Variablen** von G :

$\text{Term}(G) = \{N \in V : \exists w \in \Sigma^* \wedge N \Rightarrow^* w\}$. Ausgehend von einer ChNFG:

```
term = set({A for (A,a) in P if a in Sigma})
if term != set(): # nichtleer?
    changed = True
while changed:
    changed = False
    for (A,(B,C)) in P: # fuer alle Regeln A->BC
        if {B,C} <= term and not A in term: # "<=" = "enthalten in"
            term.add(A)
            changed = True
# term enthaelt jetzt genau die Terminierbaren
```

$L(G) \neq \emptyset$ gdw. S terminierbar, d.h. $S \in \text{Term}(G)$.

3.7-2 Das Endlichkeitsproblem: Erste Lösung

Benutze die minimale Pumping-Zahl n

Fakt $\exists w \in L(G) : n \leq |w| < 2n$ gdw. $|L(G)|$ ist unendlich.

- Richtung \Rightarrow ist offensichtlich
- Rückrichtung: Sei $z \in L$ Wort minimaler Länge $\ell \geq n$. Wäre $\ell \geq 2n$, dann könnte man gemäß P.L. zerlegen $z = uvwxy$ und $z' = uwy \in L$. z' hat jedoch Länge $\geq n$ und ist kürzer als z im Widerspruch zur Minimalität

Also braucht man „nur“ alle Wortprobleme $w \in L(G)$ für $n \leq |w| < 2n$ entscheiden. Sehr ineffizient ...

Das Endlichkeitsproblem: Zweite Lösung

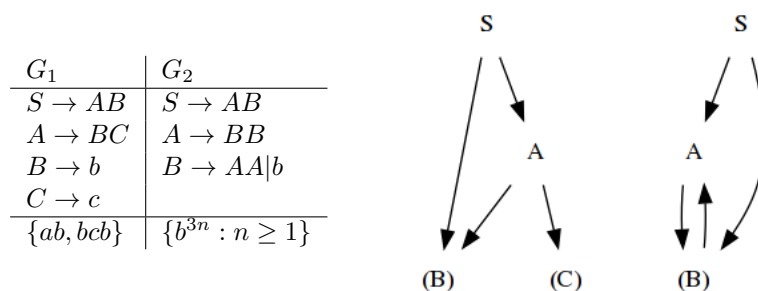
3.7-3

„Pumpen“ \Leftrightarrow Wiederholung einer Produktion (Zyklus!)

Zu gegebener ChNFG $G = (V, \Sigma, P, S)$ betrachte den *Produktionsgraphen* $G_P = (V, E)$, wobei $(A, B) \in E$ gdw. es $A \rightarrow \gamma \in P$ gibt, mit Satzform γ , die B enthält und bei der jede enthaltene Variable terminierbar ist.

Dann ist $|L(G)| = \infty$ gdw. es in G_P einen von S aus erreichbaren Zyklus gibt.

Beispiel



Unentscheidbare Probleme

3.7-4

Beispiel: Äquivalenzproblem, Schnittproblem, Mehrdeutigkeitsproblem, ...

— später mehr dazu.

Entscheidbare Varianten bei DCFL-Sprachen

Beispiel: Gegeben DPDAs D_1, D_2 , gilt $L(D_1) = L(D_2)$?

technisch sehr aufwändiger Beweis (Senizerges 1997)

4 Typ1 und Typ0-Sprachen, Turing-Maschinen

4.1 Kontextsensitive Sprachen

Kontextsensitive und monotone Produktionen

4.1-1

Kontext-sensitive Regeln sind Chomsky-Regeln der Form

$$\alpha X \beta \rightarrow \alpha \gamma \beta, \text{ wobei } \alpha, \beta, \gamma \in (V \cup \Sigma)^*, X \in V \text{ und } |\gamma| > 0$$

Beobachtung: Kontextsensitive Regeln $\ell \rightarrow r$ sind monoton, d.h. $|\ell| \leq |r|$.

Lemma

Jede Typ 1-Grammatik lässt sich äquivalent umformen in eine Grammatik aus lauter kontextsensitiven Regeln. (Beweis folgt)

Kontext-sensitive Sprachen

Typ 1-Grammatiken heißen auch **kontext-sensitiv**, die entsprechende Sprachklasse \mathcal{L}_1 wird auch mit **CSL** bezeichnet (context sensitive languages).

4.1-2 Umformung: **monoton** \Rightarrow **kontextsensitiv**

Wir gehen von einer Typ 1-Grammatik aus und benutzen die üblichen Konventionen: X, X_1, Y_1 stehen für Nonterminale, a für ein Terminalzeichen.

1. Nach *Separation von Terminalen* (siehe Paragraph 3.3-1) gibt es nur Produktionen der Gestalt $X \rightarrow a$ oder $X_1 X_2 \dots X_m \rightarrow Y_1 Y_2 \dots Y_n$.
2. Ersetze jede Produktion $X_1 X_2 \dots X_m \rightarrow Y_1 Y_2 \dots Y_n, 2 \leq m \leq n$ durch Produktionsfolge mit neuen Nonterminalen $Z_k, 1 \leq k \leq m$

$$\begin{aligned}
 X_1 X_2 \dots X_m &\rightarrow Z_1 X_2 \dots X_m \\
 Z_1 X_2 \dots X_m &\rightarrow Z_1 Z_2 X_3 \dots X_m \\
 &\vdots \\
 Z_1 Z_2 \dots Z_{m-1} X_m &\rightarrow Z_1 Z_2 \dots Z_m Y_{m+1} \dots Y_n \\
 Z_1 Z_2 \dots Z_m Y_{m+1} \dots Y_n &\rightarrow Y_1 Z_2 \dots Z_m Y_{m+1} \dots Y_n \\
 &\vdots \\
 Y_1 \dots Y_{m-1} Z_m Y_{m+1} \dots Y_n &\rightarrow Y_1 \dots Y_m Y_{m+1} \dots Y_n
 \end{aligned}$$

Folge ist nur komplett anwendbar \Rightarrow Sprache unverändert □

4.1-3 $L_{abc} = \{a^n b^n c^n | n > 0\}$ ist vom Typ 1

Die Sprache wird erzeugt durch diese Regeln (Startsymbol S)

Generatoren $S \rightarrow aSBC | aBC$

Tauscher $CB \rightarrow BC$

Terminierer $aB \rightarrow ab, bB \rightarrow bb, bC \rightarrow bc, cC \rightarrow cc$.

$S \Rightarrow aSBC \Rightarrow aaBCBC \Rightarrow aaBBCC \Rightarrow aabBCC \Rightarrow aabbCC \Rightarrow aabbC \Rightarrow aabbcc$

Korrektheit

Offenbar werden alle Wörter der Form $a^n b^n c^n, n > 0$ erzeugt. Andere nicht, denn

- ableitbare Satzformen haben gleiche Anzahl a 's, B/b 's, C/c 's,
- a 's können nur durch Generatoren entstehen, b 's, c 's nur durch Terminierer

- restloses Aufbrauchen der Nonterminale erfordert alphabetische Reihenfolge, d.h. die Tauschregeln müssen angewandt werden.

Folgerung für die Chomsky-Hierarchie

Bekanntlich gilt $L_{abc} \notin \mathcal{L}_2$, somit $\mathcal{L}_2 \subset \mathcal{L}_1$.

siehe Paragraph 3.3-6

Abschlusseigenschaften

4.1-4

Satz

Die Klasse \mathcal{L}_1 der kontextsensitiven Sprachen ist abgeschlossen unter Vereinigung, Produkt, Stern, Reflexion, nichtlöschendem Homomorphismus, Schnitt und Komplement.

Beweisideen

- Vereinigung, Produkt, Stern: wie bei kontextfreien Sprachen
- Reflexion: Übungsaufgabe
- nichtlöschender Homomorphismus: Übungsaufgabe
($h : \Sigma \rightarrow \Gamma^*$ heißt **nichtlöschend**, falls $\varepsilon \notin h(\Sigma)$)
- Komplement: nichttrivial!!, benutzt Satz von Immerman-Szelepcsényi (1987), den wir nicht an dieser Stelle behandeln
- Schnitt: folgt aus Abschluss unter Vereinigung und Komplement.

Komplexität des Wortproblems

4.1-5

Gegeben Typ 1-Grammatik $G = (V, \Sigma, P, S)$ und $x \in \Sigma^*$, gilt $x \in L(G)$?

Hier dazu noch einmal der Code aus Paragraph 3.1-2

```

1 def generated_by(x,G):          # x = Eingabewort, G = Typ 1-Grammatik
2     n = len(x)
3     (A,B) = (set(),set(G['S'])) # A = leere Menge, B = {Startsymbol}
4     while A != B:               # <=== Solange es noch Aenderungen gibt, ..
5         A = B
6         for v in B:             # <===.. bestimme Satzformen, die aus vorhandenen ..
7             for w in derive(v,G): # <== .. DIREKT ableitbar sind, ergaenze B damit ..
8                 if len(w) <= n: B.add(w) # .., falls sie nicht zu lang sind.
9     return (x in A)             # Teste, ob x dabei ist.
```

Laufzeit $O(n \cdot |P| \cdot (|V| + |\Sigma|)^{2n})$

- Zeile 6: $|B| \leq (|V| + |\Sigma|)^n$ Iterationen
- Zeile 7: auf v anwendbare Regeln bestimmen/anwenden kostet $O(n \cdot |P|)$ Schritte (String Matching), w hinzufügen nur z.B. $O(\log |B|)$
- Zeile 4: die äußere Schleife iteriert höchstens $(|V| + |\Sigma|)^n$ mal

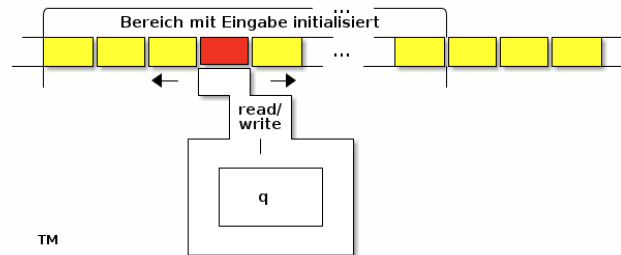
Geht es schneller?

Vermutlich nicht: das Wortproblem für Typ 1-Grammatiken ist *NP-schwer*.

Das Problem ist sogar *PSPACE-vollständig*, also möglicherweise „noch schwerer“ als NP. Diese zweistündige Vorlesung reicht leider nicht aus, diese Begriffe zu behandeln.

4.2 Turing-Maschinen

4.2-1 Mechanistische Vorstellung von Turing-Maschinen

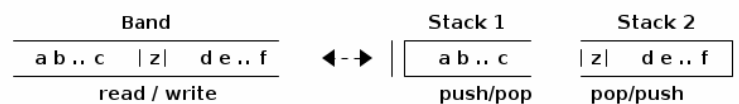


- Speicher = zweiseitig unbeschränktes Band, in Zellen unterteilt
- Schreib-/Lesekopf erfasst aktuelle Zelle (**Fokus**)
- das „Programm“ ist die Übergangstabelle
aktuelles Paar Zustand/Zeichen \mapsto mögliche Aktionen
- Aktionen = Zustandswechsel + Schreiben + Kopfbewegung
- Recheneinheit speichert *aktuellen Zustand*, steuert Ausführung von Aktionen

4.2-2 Einschub: Vergleich mit PDA

Erinnerung: $\text{PDA} \Leftarrow \text{NFA} + \text{Stack}$

Beobachtung



Das TM-Leseband ist äquivalent zu zwei Stacks.

4.2-3 Formale Definition

Eine *nichtdeterministische Turing-Maschine* $(Q, \Sigma, \Gamma, \Delta, q_0, \square, F)$ besteht aus

- Zustandsmenge Q , Startzustand $q_0 \in Q$, Eingabealphabet Σ , Menge von Endzuständen $F \subseteq Q$
- **Arbeitsalphabet** $\Gamma \supset \Sigma$ mit Sonderzeichen $\square \in \Gamma \setminus \Sigma$ (**Blank**) und $\Gamma \cap Q = \emptyset$ sowie **Überföhrungsfunktion** $\Delta : Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L, R, S\}}$

Nichtdeterministische TM erlauben ebenso wie NFA und PDA auch \emptyset als Ergebnis Die TM ist **deterministisch (DTM)**, falls $\forall q \in Q \forall \gamma \in \Gamma : |\Delta(q, \gamma)| \leq 1$.

Δ kann dann *partiell* angegeben werden durch $\delta : Q \times \Gamma \rightrightarrows Q \times \Gamma \times \{L, R, S\}$ mit

$$\Delta(q, \gamma) = \{\delta(q, \gamma)\} \text{ (sofern } \Delta(q, \gamma) \neq \emptyset \text{)}.$$

Arbeitsweise einer Turing-Maschine

4.2-4

Δ -Tripel $(q', \gamma, mv) \in Q \times \Gamma \times \{L, R, S\}$ beschreibt eine ausführbare Aktionen:

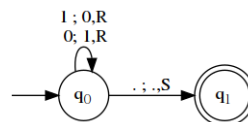
- (1) in Zustand q' wechseln,
- (2) gelesenes Zeichen durch γ ersetzen und
- (3) Schreib-/Lesekopf gemäß mv positionieren:
LINKS/RECHTS/STEHEN.

Beispiel: Bit-Flipper

$T_F = (\{q_0, q_1\}, \{0, 1\}, \{0, 1, \square\}, q_0, \square, \Delta, \{q_1\})$ mit partieller Übergangsfunktion

δ	0	1	\square
q_0	$(q_0, 1, R)$	$(q_0, 0, R)$	(q_1, \square, S)

```
TMflip = {'Q': {'q0', 'q1'},
'Sigma': {'0', '1'},
'Gamma': {'0', '1', '.'},
'Delta': {
('q0', '0'): {'q0', '1', 'R'},
('q0', '1'): {'q0', '0', 'R'},
('q0', '.'): {'q1', '.', 'S'}
},
'q0': 'q0',
'B': '.', # . = Blank
'F': {'q1'}}
```



bequemer mit den üblichen Konventionen und zusätzlich $.$ als einzig möglichem Blank-Symbol

```
TMflip = md2mc('''TM
Iq0 : 0; 1,R
Iq0 : 1; 0,R -> Iq0
Iq0 : .; .,S -> Fq1
''')
# oder auch
TMflip = md2mc('''TM
Iq0 : 0; 1,R | 1; 0,R -> Iq0
Iq0 : .; .,S -> Fq1
''')
```

Konfigurationen beschreiben globalen Berechnungszustand

4.2-5

- bei PDAs: (Zustand, Restwort, Kellerwort)
- bei TMs: (Zustand, relevanter Bandinhalt, Kopfposition darin)

Definition

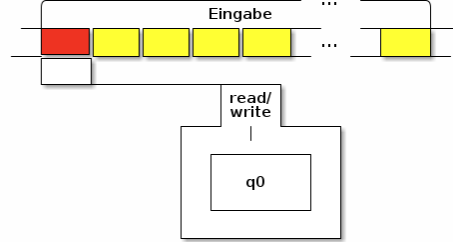
Eine **Konfiguration** von $(Q, \Sigma, \Gamma, \Delta, q_0, \square, F)$ ist ein Wort $k = \alpha q \beta \in \Gamma^* Q \Gamma^*$.

Interpretation:

- q = aktueller Zustand,
- $\alpha\beta$ = relevanter Bandinhalt (**Inschrift**)
- Fokus = erstes Zeichen von β .

Die **Startkonfiguration** auf Eingabe x ist definiert durch

$$k_0(x) = \begin{cases} q_0 \square & \text{falls } x = \varepsilon \\ q_0 x & \text{sonst.} \end{cases}$$



4.2-6 Konfigurationsübergänge

Sei M eine TM. Auf ihrer Konfigurationsmenge $\Gamma^* Q \Gamma^*$ wird die **Übergangsrelation** \vdash_M definiert: falls $k \vdash_M k'$, so heißt k' eine **Nachfolgekongfiguration** von k .

Definition

Folgende Konfigurationen stehen in Relation:

$$\alpha_1 \dots \alpha_m q \beta_1 \dots \beta_n \vdash_M \begin{cases} \alpha_1 \dots \alpha_m q' \gamma \beta_2 \dots \beta_n, & \text{falls } \Delta(q, \beta_1) \ni (q', \gamma, S), m \geq 0, n \geq 1 \\ \alpha_1 \dots \alpha_m \gamma q' \beta_2 \dots \beta_n, & \text{falls } \Delta(q, \beta_1) \ni (q', \gamma, R), m \geq 0, n \geq 2 \\ \alpha_1 \dots \alpha_{m-1} q' \alpha_m \gamma \beta_2 \dots \beta_n, & \text{falls } \Delta(q, \beta_1) \ni (q', \gamma, L), m \geq 1, n \geq 1 \end{cases}$$

und außerdem die folgenden (**Blank-Ergänzung an Inschrift-Enden**):

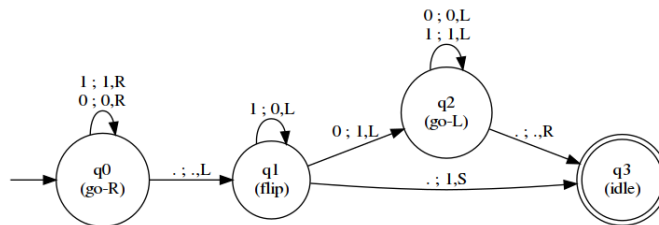
$$\begin{aligned} \alpha_1 \dots \alpha_m q \beta_1 \vdash_M \alpha_1 \dots \alpha_m \gamma q' \square, & \text{falls } \delta(q, \beta_1) \ni (q', \gamma, R) \\ q \beta_1 \dots \beta_n \vdash_M q' \square \gamma \beta_2 \dots \beta_n, & \text{falls } \delta(q, \beta_1) \ni (q', \gamma, L) \end{aligned}$$

Erklärung Wenn M jenseits des beschriebenen Bandinhalts eine Zelle *erstmalig* betritt, ist sie leer. Diese Information wird in der Konfiguration gespeichert.

4.2-7 Beispiel: Nachfolgerberechnung

gegeben $x = \text{bin}(n)$

gesucht $\text{bin}(n+1)$



Konfigurationsfolge für $x = 101$

$$q_0 101 \vdash 1 q_0 01 \vdash 10 q_0 1 \vdash 101 q_0 \square \vdash 10 q_1 1 \square \vdash 1 q_1 00 \square \vdash q_2 110 \vdash q_2 \square 110 \square \vdash \square q_3 110 \square$$

Konfigurationsfolge für $x = 11$

$q_0 11 \vdash 1q_0 1 \vdash 11q_0 \square \vdash 1q_1 1 \square \vdash q_1 10 \square \vdash q_1 \square 00 \square \vdash q_3 100 \square$

Turing-Akzeptoren

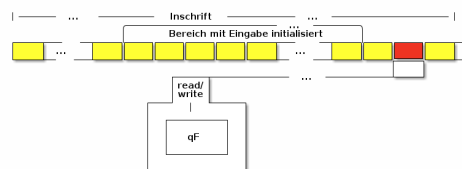
4.2-8

Turing-Akzeptor = „als Spracherkenner betriebene“ TM

$M = (Q, \Sigma, \Gamma, \Delta, q_0, \square, F)$

Definition

$\alpha q \beta \in \Gamma^* Q \Gamma^*$ mit $q \in F$ heißt **Endkonfiguration** (Sammelbezeichnung: k_F).



Von M **erkannte Sprache**: $L(M) = \{w \in \Sigma^* : k_0(w) \vdash_M^* k_F\}$.

Nützliche Notation für oft gebrauchte Formulierung

$M(w)$ steht im folgenden für „ M , gestartet auf Eingabe w “. Also

$$L(M) = \{w \in \Sigma^* : M(w) \text{ kann eine Endkonfiguration erreichen}\}.$$

Beispiel: DTM für die Sprache $\{w\#w : w \in \{0,1\}^*\}$

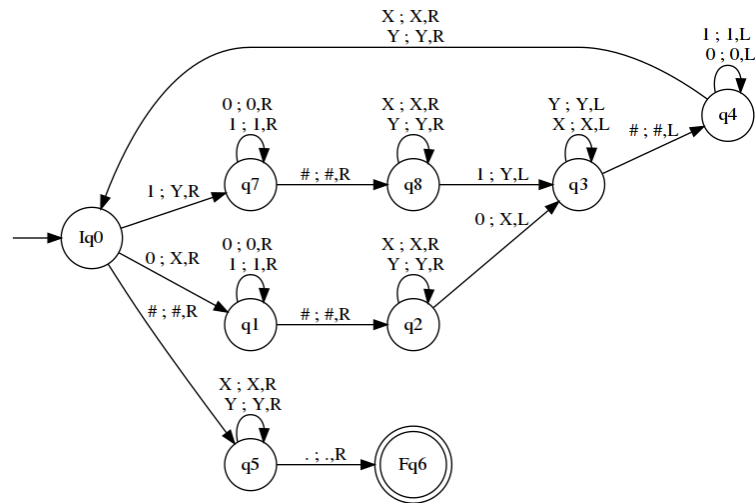
4.2-9

Von links beginnend einzelne Zeichen mit rotem oder blauem Stift abstreichen. Wir unterscheiden linke und rechte Zeichen, je nachdem auf welcher Seite von $\#$ sie stehen.

Ist das nächste zu verarbeitende linke Zeichen eine 0 bzw. 1, so wird es durch X bzw. Y ersetzt (rot bzw. blau abstreichen). Das nächste unverarbeitete rechte Zeichen muss dann ebenfalls 0 bzw. 1 sein und wird durch X bzw. Y ersetzt. Ist es ein anderes Zeichen (oder Blank), so wird verworfen.

Nach dem Abstreichen eines Paares wird zurückgewandert bis zum ersten linken X bzw. Y. Das Zeichen rechts daneben ist das nächste zu verarbeitende. Falls dies das Mittensymbol $\#$ ist, so muss auf der rechten Seite hinter dem letzten X oder Y ein Blank stehen, sonst wird verworfen.

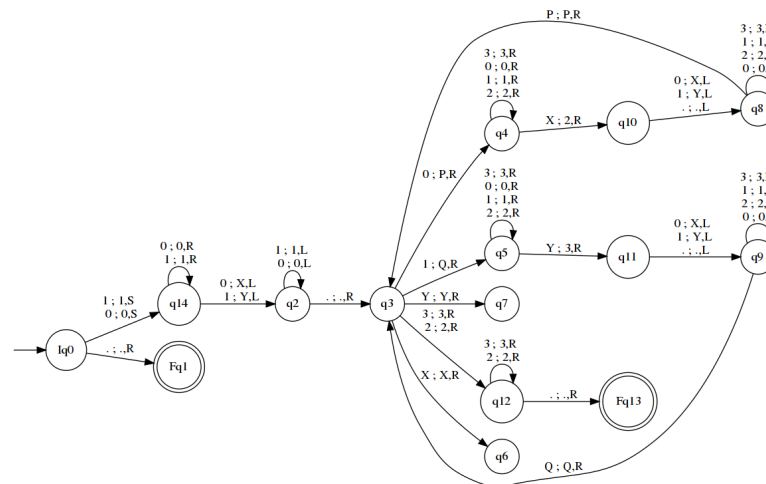
Bemerkung: Die Unterscheidung rot/blau X/Y ist nicht wesentlich, denn das könnte im Zustand gespeichert werden.



4.2-10 Beispiel: TM für die Sprache $\{ww : w \in \{0,1\}^*\}$

Idee

Wie zuvor, nur muss die Mitte geraten werden \Rightarrow Nichtdeterminismus



4.2-11 Nichtdeterministische Akzeptoren können durch deterministische simuliert werden!

Gegeben TM M und Eingabe w

Gefragt gilt $k_0(w) \vdash_M^* k_F$?

Idee

- simulierte *systematisch* alle möglichen Berechnungen (Konfigurationsfolgen) von $M(w)$ der Längen 0, 1, 2, ... und bis eine von ihnen zu einer Endkonfiguration führt

- die Simulation terminiert genau dann, wenn $M(w)$ akzeptiert

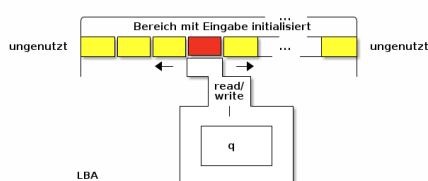
4.3 Akzeptor-Charakterisierungen von Typ 1- und Typ 0-Sprachen

Linear beschränkte Automaten

4.3-1

LBA = nichtdeterministische TM, die den „Eingabebereich“ nie verlässt“

Bemerkung: Die formale Definition sorgt wie folgt für das Erkennen des Inskrift-Endes. Die übliche Startkonfiguration q_0x auf Eingabe x wird durch eine Version ersetzt, in der das letzte Zeichen entsprechend markiert ist. Beim Inskriftanfang (auf dem anfangs ja der Kopf steht) kann die Maschine selbst für die Markierung sorgen. Das erfordert eine Verdopplung des Eingabealphabets: für jedes Zeichen muss auch die markierte Variante aufgenommen werden.



Genauer (Vorbereitung):

$M = (Q, \Sigma, \Gamma, \Delta, q_0, \square, F)$ sei TM mit $\Gamma \supseteq \Sigma \cup \hat{\Sigma}$, wobei

$$\hat{\Sigma} = \{\hat{\sigma} : \sigma \in \Sigma\}.$$

Aus $x \in \Sigma^*$ entsteht \hat{x} , indem letztes Zeichen σ durch $\hat{\sigma}$ ersetzt wird.

Definition

Die **Ende-markierte Startkonfiguration** von $M(x)$ ist $\hat{k}_0(x) = q_0\hat{x} \in \Gamma^*Q\Gamma^*$.

M heißt **linear beschränkter Automat** (LBA), falls für alle $x \in \Sigma^*$:

$$\text{aus } \hat{k}_0(x) \vdash_M^* \alpha q \beta \text{ folgt } |\alpha\beta| = |x|.$$

- die Maschine kann selbst das Anfangszeichen markieren und
- anhand der Markierung ist erkennbar, ob linkes/rechtes Inskriptende erreicht ist

Beispiel

Der nichtdeterministische Erkennen für $\{ww : w \in \{0,1\}^*\}$ (siehe Paragraph 4.2-10) kann als LBA aufgefasst werden.

Charakterisierungssätze

4.3-2

Satz von Kuroda

Die von LBAs erkannten Sprachen sind genau die Typ 1-Sprachen (kontextsensitive Sprachen).

Satz (mit fast gleichem Beweis)

Die von nichtdeterministischen Turing-Akzeptoren erkannten Sprachen sind genau die Typ 0-Sprachen.

4.3-3 Beweis Teil 1: Grammatik $G \rightsquigarrow$ Turing-Akzeptor

Sei $G = (V, \Sigma, P, S)$ eine Chomsky-Grammatik.

Turing-Akzeptor M_G

Idee errate **Reduktion** (!) zum Startsymbol $S \Leftarrow^* w$

Initialisiere aktuelle Inschrift $I :=$ Eingabewort w

Iteriere solange $I \neq S$

- wähle irgendeine Produktion $u \rightarrow v \in P$
- suche nichtdeterministisch Zerlegung der Inschrift: $I = \pi v \sigma$
 - falls gefunden, reduziere: $\pi u \sigma \Leftarrow I$

Terminierte gehe in Endkonfiguration

Korrektheit

$$\begin{aligned} w \in L(G) &\iff \exists \text{ Ableitungssequenz } S \Rightarrow \dots \Rightarrow w \\ &\iff \exists \text{ Reduktion } S \Leftarrow \dots \Leftarrow w \\ &\iff w \in L(M) \end{aligned}$$

4.3-4 Beweis Teil 1 (Ende)

- Obiger Algorithmus kann in TM-Übergangstabelle umformuliert werden (kleine technische Hürde: unterschiedliche Längen $|u|, |v|$).
- Damit folgt die Aussage für Typ 0-Sprachen
- Ist die Grammatik kontextsensitiv, so ist sie insbesondere monoton.
- Die Ersetzung kann dann so gestaltet werden, dass der Akzeptor “nicht den Eingabebereich verlässt”, also ein LBA ist. (siehe dazu die Bemerkung in Paragraph 4.3-1 im Skript)

Damit folgt die Aussage für Typ 1-Sprachen. □

4.3-5 Beweis Teil 2: LBA \rightsquigarrow Grammatik

Zu jedem LBA M existiert eine Typ 1-Grammatik $G = (V, \Sigma, P, S)$ mit

$$\hat{k}_0(w) \vdash_M^* k_F \text{ genau dann, wenn } w \in L(G).$$

Beweis in folgenden Schritten:

Codierung codiere Konfigurationen k durch Satzformen \bar{k}

Simulation entwirf kontextsensitive bzw. monotone Produktionen, die Rechenschritte simulieren

Konservierung ergänze Produktionen, um das Eingabewort über alle Ableitungsschritte zu erhalten

„**Finale**“ folgere Äquivalenz von G und M

Beweis Teil 2: Codierung

4.3-6

Konfiguration $k = \alpha q \beta \in \Gamma^* Q \Gamma^*$

Details

- **Problem:** k kann Länge $n + 1$ haben ($n = |w|$), Grammatik darf aber nur Satzformen der Länge $\leq n$ verwenden!
- **Lösung:** q und Anfangszeichen B von β wird zusammengefasst zu „Superzeichen“ B_q (bzw. (q, B)) \Rightarrow Alphabeterweiterung auf $\Gamma_Q := \Gamma \cup (Q \times \Gamma)$
Beispiel: Aus Konfiguration $k = AqBCD$ mit Länge 5 wird die Satzform $\bar{k} := A(q, B)CD$ mit Länge 4.

Beweis Teil 2: Simulation

4.3-7

- Δ -Übergänge von M entsprechen Produktionen der Grammatik, die je nach Bewegungsrichtung anders aussehen:
 - $(q', A', S) \in \Delta(q, A) \Rightarrow$ Produktion $(q, A) \rightarrow (q', A')$
 - $(q', A', R) \in \Delta(q, A) \Rightarrow$ für jedes $B \in \Gamma$ eine Produktion $(q, A)B \rightarrow A'(q', B)$
 - $(q', A', L) \in \Delta(q, A) \Rightarrow$ für jedes $B \in \Gamma$ eine Produktion $B(q, A) \rightarrow (q', B)A'$
- Diese kontextsensitive(!) Produktionsmenge sei P' . Offensichtlich gilt

$$k_1 \vdash_M^* k_2 \iff \bar{k}_1 \Rightarrow_{P'}^* \bar{k}_2.$$

Beweis Teil 2: Konservierung

4.3-8

- **Problem:** die Ableitung soll nicht die zur Endkonfiguration passende Satzform generieren, sondern das Eingabewort
- **Lösung:** blähe Zeichen zu Paaren auf, die wir uns wie Dominosteine vorstellen (erste/zweite Komponente = Ober-/Unterseite); aneinandergelegt sind dann zu lesen
 - oben: eine codierte Konfiguration $\in \Gamma_Q^*$ und
 - unten: das Eingabewort $\in \Sigma^*$also: $V = \Gamma_Q \times \Sigma$
- System von Produktionen besteht dann aus
 - *Anfangsproduktionen:* generieren Startkonfiguration + Eingabewort
 - *erweiterte Produktionen:* Produktionen von P' (Oberseite) erweitert um die richtige Behandlung der Unterseite
 - *Schlussproduktionen:* extrahieren Eingabe aus Endkonfiguration UND Eingabe

4.3-9 Zwischenübersicht: Die Grammatik G

- Terminale (= von P' verwendete Zeichen):

$$\Gamma_Q := \Gamma \cup (Q \times \Gamma)$$

- Nonterminale (mit Startsymbol S):

$$V := \{S, T\} \cup (\Gamma_Q \times \Sigma)$$

- Produktionsregeln:

- Anfangsregeln - für alle $\sigma \in \Sigma$
 $S \rightarrow T(\hat{\sigma}, \sigma), T \rightarrow T(\sigma, \sigma) | ((q_0, \sigma), \sigma)$
 $(\hat{\sigma} = \text{„}\sigma \text{ mit Endekennzeichnung“})$
- erweiterte Produktionen - für alle $\sigma, \tau \in \Sigma$:
 $(X, \sigma)(Y, \tau) \rightarrow (X', \sigma)(Y', \tau)$ falls $XY \rightarrow X'Y' \in P'$
 $(X, \sigma) \rightarrow (X', \sigma)$ falls $X \rightarrow X' \in P'$
- Schlussregeln - für alle $q \in F, A \in \Gamma, \sigma \in \Sigma$:
 $(A, \sigma) \rightarrow \sigma$ und $((q, A), \sigma) \rightarrow \sigma$

4.3-10 Beweis Teil 2: „Finale“

folgende Aussagen sind äquivalent:

1. $w_1 w_2 \dots w_n \in L_M$
2. \exists Folge $q_0 w_1 w_2 \dots \hat{w}_n =: \hat{k}_0(w) \vdash_M k_1 \vdash_M \dots$ von Konfigurationsübergängen zu k_F
3. \exists G -Ableitung, die eine Satzform $\in (\Gamma_Q \times \Sigma)^*$ generiert, deren Oberseite eine Satzform $\overline{k_F}$ ergeben, die eine Endkonfiguration k_F codiert und deren Unterseite w ergeben.
(Mit den Schlussregeln ist aus dieser Satzform w ableitbar.)
4. $w_1 w_2 \dots w_n$ gehört zu der durch G generierten Sprache $L(G)$ □

„ergeben ...“ bedeutet jeweils: „sind hintereinander geschrieben (oder als Dominosteine aneinandergelegt) gleich ...“

4.3-11 Das Analogon für Typ 0-Sprachen

Zu jedem Turing-Akzeptor M existiert Grammatik $G = (V, \Sigma, P, S)$ mit

$$w \in L(G) \text{ genau dann, wenn } k_0(w) \vdash_M^* k_F.$$

Unterscheidung zwischen $k_0(w)$ und $\hat{k}_0(w)$ ist nur bei LBA wichtig

Beweis

- analog zu entsprechendem Beweis für LBA
- **technisches Problem:** die Grammatik-Ableitung von $w \in L_M$ kann auch Satzformen mit Länge $> |w|$ enthalten!
- um daraus w abzuleiten, werden geeignete (in kontext-sensitiven Grammatiken allerdings verbotene) ε -Produktionen als Abschlussregeln zugelassen, um Teile der Satzform außerhalb des Eingabebereichs zu löschen

□

4.4 Einschub: Einige technische Vorbereitungen

Es werden einige noch mehrfach benötigte Konstruktionen vorgestellt.

[Funktionen als Zuordnung](#)

4.4-1

Erinnerung

Eine totale Funktion $f : X \rightarrow Y$ ist eine Teilmenge $f \subseteq X \times Y$, so dass für jedes $x \in X$ ein y existiert mit $(x, y) \in f$ (Notation $f : x \mapsto y$)

Beispiel: $f : x \mapsto x + 2$, $add : (a, b) \mapsto a + b$

λ -Ausdrücke

Wir betrachten die Zuordnung als mathematisches Objekt, das keinen Namen hat, sondern für sich steht.

Kennzeichnung: $\lambda x. x + 2$, $\lambda ab. a + b$

Der λ -Kalkül ist nicht klausurrelevant. Die Grundidee ist trotzdem leicht zu verstehen und vor allem ungemein praktisch in unseren Python-Codeschnipseln.

Lambda-Ausdrücke in Python

```
1 lambda x: x+2          # "anonyme" Zuordnung: dem x wird x+2 zugeordnet
2 lambda a,b: a+b        # eine andere Zuordnung
3 result = (lambda x: x+2)(5) # Anwendung der Zuordnung auf eine Zahl
4 f = lambda x: x+2      # die Zuordnung unter dem Namen f speichern
5 result = f(5)          # gleiche Wirkung wie in Zeile 3
```

Vergleich: Dictionaries definieren nur endlich viele Zuordnungspaare.

Anwendung: Positive natürliche Zahlen \Leftrightarrow Bitstrings

4.4-2

Erinnerung: Bitstrings in *längen-lexikographischer Aufzählung* (siehe Paragraph 1.3-8):

Bitstring b	ε	0	1	00	01	10	11	000	001	...
Ordnungszahl n	1	2	3	4	5	6	7	8	9	...

Dies ist eine *bijektive Abbildung* $\{0,1\}^* \Leftrightarrow \mathbb{N}_+$. Die Abbildung und ihre Umkehrung als Lambda-Ausdrücke formuliert:

`n2b` entspricht genau der in Kapitel 1 eingeführten Funktion `ntbbitstring`).

```
n2b = lambda n: bin(n)[3:] # number-to-bits
b2n = lambda s: int('1'+s,2) # bits-to-number
```

Vergleich: Funktionsdefinition mit def

```
def n2b(n):
    return bin(n)[3:]
def b2n(s):
    return int('1'+s,2)
```

Die Cantorsche Paarungsfunktion

4.4-3

Eine bijektive Zuordnung $A \times B \rightarrow C$ heißt **Paarung**: Paare werden durch „normale Elemente“ *codiert*.

Beispiel

Die **Cantor-Paarung** π codiert jedes Zahlenpaar durch seine Nummer bei \nearrow -diagonaler Nummerierung:

$x \backslash y$	0	1	2	3	...
0	0	2	5	9	...
1	1	4	8	13	...
2	3	7	12	18	...
3	6	11	17	24	...
...

$$\begin{aligned}\pi(x, y) &= y + \sum_{i=0}^{x+y} i \\ &= y + \frac{1}{2}(x+y)(x+y+1) \\ &= y + \binom{x+y+1}{2}\end{aligned}$$

Python-Implementierung

```
pi = lambda x,y: y + (x+y)*(x+y+1)//2
```

// anstelle / ist wichtig, denn bei Python ergibt Division standardmäßig einen Gleitkommawert.

Bemerkung

Offenbar definiert dies auch eine Bijektion $\mathbb{N}_+ \rightarrow \mathbb{N}_+ \times \mathbb{N}_+$ — wichtig bei Kombination mit $\{0, 1\}^* \simeq \mathbb{N}_+$ -Bijektion.

4.4-4 Cantor-Projektionen = Umkehrungen der Cantor-Paarung

$$\pi(x, y) =: n \stackrel{\pi_L}{\mapsto} x, n \stackrel{\pi_R}{\mapsto} y.$$

$x \backslash y$	0	1	2	3	...
0	0	2	5	9	...
1	1	4	8	13	...
2	3	7	12	18	...
3	6	11	17	24	...
...

Spalte 0 enthält die „Dreieckszahlen“

$$\Delta = \{0, 1, 3, 6, 10, \dots, \binom{x}{2}, \dots\}.$$

Beobachtung

$y = n - t_{\max}$ mit

$$t_{\max} = \max\{t \in \Delta : t \leq n\}.$$

Ähnlich wird x bestimmt.

Python-Variante (hier ohne Korrektheitsbeweis)

```
import math
trirt = lambda x: math.floor((math.sqrt(8*x+1)-1)/2) # trirt = triangular root :-)
piL = lambda z: trirt(z) - (z - pi(trirt(z),0))
piR = lambda z: z - pi(trirt(z),0)
```

4.4-5 Codierung/Decodierung von Zahlentupeln

Zahlentupel als Zahlen codieren durch iterierte Anwendung der Cantor-Paarung:

$$\langle n_1, n_2, \dots, n_k \rangle := \pi(n_1, \pi(n_2, \dots, \pi(n_k, 0) \dots)).$$

Variante 1: $k > 0$ fixiert (Bijektion zwischen \mathbb{N} und \mathbb{N}^k)

Aus der Codierung $n := \langle n_1, n_2, \dots, n_k \rangle$ die Komponenten n_1, n_2, \dots, n_k berechnen:

n_1 : Wegen $n = \pi(n_1, ?)$ ist $n_1 = \pi_L(n)$

n_2 : Wegen $n = \pi(n_1, \pi(n_2, ?))$ ist $n_2 = \pi_L(\pi_R(n))$

...

n_k : Wegen $n = \pi(n_1, \pi(n_2, \dots, \pi(n_k, 0) \dots))$ ist $n_k = \pi_L(\underbrace{\pi_R(\pi_R(\dots \pi_R(n) \dots))}_{k-1})$

```
# decode: Zahl |--> k-Tupel, encode: Tupel |--> Zahl
dec=lambda n,k: (piL(n),) if k==1 else tuple(list(dec(piR(n),k-1))+[piL(n)])
enc=lambda tup: pi(tup[-1],0) if len(tup)==1 else pi(tup[-1],enc(tup[:-1]))
```

Variante 2: $k > 0$ variabel (Bijektion zwischen \mathbb{N} und \mathbb{N}^+)

```
t2n=lambda t: enc(t) # tupel-to-number (wie oben)
n2t=lambda n: dec(n,piR(n)+1) # number-to-tupel: Tupellaenge k=piR(n)+1 > 0
```

4.5 Rekursiv aufzählbare Sprachen

Eine Typ-0 Grammatik

4.5-1

Typ 0-Sprachen sind die durch uneingeschränkte Grammatiken $G = (V, \Sigma, P, S)$ erzeugten Sprachen.

Beispiel (Übungsaufgabe)

Die Sprache $L_{a2n} = \{a^{2^n} | n > 0\}$ wird erzeugt durch die Typ 0-Grammatik

$$S \rightarrow SD | La, aD \rightarrow Daa, LD \rightarrow L, L \rightarrow \varepsilon$$

siehe Jupyter-Notebook

Das Wortproblem für Typ-0-Sprachen entscheiden?

4.5-2

systematisch 0, 1, 2, ... Schritte aus S ableiten bis Ableitung für x gefunden:

- Im Fall $x \in L(G)$ wird dies bestätigt und richtig entschieden.
- Im Fall $x \notin L(G)$ wird *nicht* entschieden: es gibt eine Endlosschleife (kein „Abschneiden“ wegen zu großer Wortlängen möglich)!

```
1 def generated_by(x,G): # x = Eingabewort, G = >>>Typ 0-Grammatik<<<
2   (A,B) = (set(),set(G['S'])) # A = leere Menge, B = {Startsymbol}
3   while A != B: # <=== Solange es noch Aenderungen gibt, ..
4     A = B
5     for v in B: # <===.. bestimme Satzformen, die aus vorhandenen ..
6       for w in derive(v,G): # <== .. DIREKT ableitbar sind, ergaenze B damit ..
7         B.add(w)
8   if x in B:
9     return True
```

Damit kann man das Wortproblem „nur zur Hälfte“ entscheiden — es ist *semi-entscheidbar*.

4.5-3 Rekursive Aufzählbarkeit — eine Charakterisierung für Typ0

Definition

$L \subseteq \Sigma^*$ heißt **rekursiv aufzählbar** falls $L = \emptyset$ oder es gibt eine *berechenbare* Funktion $f : \mathbb{N}_+ \rightarrow \Sigma^*$ gibt mit

$$L = \{f(1), f(2), f(3), f(4), \dots\}$$

(Mehrfachnennung erlaubt). Die Klasse dieser Sprachen wird mit **RE** bezeichnet (bzw. RE ohne Alphabetfestlegung) für *recursively enumerable*.

Satz

$\mathcal{L}_0 = \text{RE}$.

Beispiel (Beweis später)

Das *Halteproblem für Turing-Maschinen* — diese Sprache untersuchen wir im nächsten Kapitel — ist rekursiv aufzählbar aber nicht kontextsensitiv. Konsequenz für die Chomsky-Hierarchie: $\mathcal{L}_1 \subset \mathcal{L}_0$.

4.5-4 (Teil-)Beweis des Satzes für $\Sigma = \{0,1\}$

Beweis (Teil 1): Wenn $L \in \mathcal{L}_0$, so ist L rekursiv aufzählbar.

Die Aussage trifft offenbar zu für $L = \emptyset$. Sei also $L = L(G)$ Typ 0-Sprache und $\text{JOKER} \in L$ ein beliebiges Wort.

Folgende Funktion ist berechenbar durch einen „Semi-Entscheider mit Timer“ siehe Paragraph 4.5-2:

```
# systematisch Ableitungssequenzen bis Laenge t probieren, in "Pseudo Python":  
g_timed(z,t)=lambda z,t: True if "S ==>* z in bis zu t Schritten" else False
```

Somit ist auch folgende Funktion berechenbar:

```
def f(n):  
    (s,t) = (piL(n),piR(n)) # decodiere n in Zahlenpaar (Cantor-Projektionen)  
    x = n2b(s)              # interpretiere s als Bitstring  
    if g_timed(x,t): return x  
    else: return JOKER      # kann beliebig oft genannt werden
```

f zählt L rekursiv auf: es werden *nur* Wörter aus L geliefert und zwar *jedes*, da jede Kombination aus Bitstring und „Rechenzeit“ versucht wird. \square

Der Beweis der Rückrichtung wird auf später verschoben.

4.5-5 Abschlusseigenschaften

Das Wort „rekursiv“ hat an dieser Stelle erst mal nichts mit Rekursion vs. Iteration zu tun.

Satz

$\mathcal{L}_0 = \text{RE}$ ist nicht abgeschlossen unter Komplement, aber unter Vereinigung, Verkettung, Stern, Homomorphismus und Schnitt.

Zum Beweis: (1) Komplement und Homomorphismus

- Abgeschlossenheit unter Homomorphismus: Übung
- Nichtabgeschlossenheit unter Komplement: später
- auch später: die Rechtfertigung für die Gleichsetzung
„berechenbar“ = „kann in Python implementiert werden“

Naheliegende Idee für die anderen Abgeschlossenheiten

- seien für die Ausgangssprachen rekursive Aufzählungsfunktionen $\mathbb{N}_+ \rightarrow \{0,1\}^*$ gegeben
- konstruiere daraus Aufzählungen für Vereinigung, Verkettung, Stern, Schnitt
- nötig ist das aber nur bei nichtleerem Ergebnis — sonst folgt rekursive Aufzählbarkeit nach Definition

Beweis: (2) Vereinigung, Verkettung und Stern

4.5-6

Seien f und g Python-Funktionen, die $L = L_1, L_2 \subseteq \Sigma^*$ rekursiv aufzählen. Folgende zählen dann $L_1 \cup L_2$ und $L_1 \cdot L_2$ rekursiv auf ...

```
vereinigung = lambda i: f(i/2) if i%2 == 0 else g((i+1)/2)
verkettung   = lambda i: f(piR(i))+g(piL(i)) # Stringverkettung mit Operator +
```

... und diese L^* :

```
def stern(i):
    if i==0:
        return ''
    else: # betrachte i-1 = <j1,...,jk> mit k>0
        tupl = n2t(i-1) # i-1 als Zahl Tupel <j1, ...,jk> decodieren und mit ..
        strs = [f(j) for j in tupl] #.. number-to-bitstring in String-Liste..
        return ''.join(strs)      #.. umsetzen und deren Verkettung liefern
```

Beweis des Satzes: (3) Schnitt

4.5-7

Sei JOKER ein Wort aus dem Schnitt. Folgende Funktion liefert offenbar nur Elemente aus $L_1 \cap L_2$:

```
def schnitt(i): # betrachte i = <n,t1,t2>
    (n,t1,t2) = dec(i,3) # decodiere als Tripel
    x = n2b(n) # einen String bestimmen mit number-to-bitstring

    # ueberpruefe maximal t1 bzw. t2 Ableitungsschritte ..
    # .. anhand von Typ0-Grammatiken fuer L1 und L2:
    if g1_timed(x,t1) and g2_timed(x,t2):
        return x
    else:
        return JOKER
```


Andererseits tritt *jedes* Element aus dem Schnitt auf: Falls $x \in L_1 \cap L_2$, so $\text{g1_timed}(x, t_1) == \text{g2_timed}(x, t_2) == \text{True}$ für gewisse t_1, t_2 . Entsprechend gilt $\text{schnitt}(i) == x$ für $n = \text{b2n}(x)$ und $i = \text{enc}(n, t_1, t_2)$.

Mit anderen Worten: `schnitt` zählt $L_1 \cap L_2$ rekursiv auf. □

4.6 Überblick/Zusammenfassung

4.6-1 Die Chomsky-Hierarchie

Grammatiken

- beschreiben Wortmengen durch Ersetzungssysteme
- Typen 0, 1, 2, 3 mit immer restriktiveren Regeln
- entsprechend Chomsky-Klassen $\mathcal{L}_i, i = 0, 1, 2, 3$

Chomsky-Hierarchie

- die Hierarchie ist echt: $\mathcal{L}_3 \subset \mathcal{L}_2 \subset \mathcal{L}_1 \subset \mathcal{L}_0$
- $\{a^n b^n : n \in \mathbb{N}\} \in \mathcal{L}_2 \setminus \mathcal{L}_3$ — Pumping-Lemma 1
- $\{a^n b^n c^n : n \in \mathbb{N}\} \in \mathcal{L}_1 \setminus \mathcal{L}_2$ — Pumping-Lemma 2
- TM-Halteproblem $\in \mathcal{L}_0 \setminus \mathcal{L}_1$ (Erklärung+Beweis folgt)

Es gibt $L \notin \mathcal{L}_0$

- $\{(M_1, M_2) : M_1, M_2 \text{ sind Turing-Akzeptoren und } L(M_1) \neq L(M_2)\}$ ist keine Typ 0-Sprache
- hier kein Beweis

4.6-2 Ein einfacher nichtkonstruktiver Beweis

Beweis(skizze) durch **Diagonalisierung**

- angenommen *jedes* $L \subseteq \{0, 1\}^*$ wird durch Typ 0-Grammatik generiert
- jede Grammatik G kann z.B. als Python-Dictionary spezifiziert, das Dictionary wiederum durch einen Bitstring g codiert werden
- sei g_1, g_2, \dots die numerische Aufzählung der (codierten) Typ 0-Grammatiken G_1, G_2, \dots über $\{0, 1\}$ und seien $L_1 = L(G_1), L_2 = L(G_2), \dots \subseteq \{0, 1\}^*$ deren erzeugte Sprachen
- weiter sei $w_1 = \varepsilon, w_2 = 0, w_3 = 1, \dots$ die numerische Aufzählung der Binärstrings
- die Sprache $L = \{w_n : w_n \notin L_n\} \subseteq \{0, 1\}^*$ kann nicht in der Liste sein, ist also keine Typ 0-Sprache

4.6-3 Struktur-Charakterisierung von Typ i -Sprachen

Klasse	charakterisierende Eigenschaft	Struktur
\mathcal{L}_3	regulär (REG)	reg. Ausdr. ⁷ : r
\mathcal{L}_2	kontextfrei (CFL)	$h(r \cap D_n)$ ⁸
\mathcal{L}_1	kontextsensitiv (CSL)	allgemeiner, h nichtlöschend ⁹
\mathcal{L}_0	rekursiv aufzählbar (RE)	allgemeiner, h beliebig

Akzeptor-Charakterisierung der Sprachklassen

4.6-4

Grammatik	Sprachklasse	Akzeptor
Typ 3	REG	DFA, NFA
LR(k)	det.-kontextfrei (DCFL)	DPDA
Typ 2	CFL	PDA
Typ 1	CSL	LBA
Typ 0	RE	TM, DTM

Determinismus und Nichtdeterminismus

4.6-5

nichtdeterministisch	deterministisch	äquivalent	Bemerkung
NFA	DFA	ja	Satz von Rabin&Scott (1959)
PDA	DPDA	nein	Gegenbeispiel: Palindrome
LBA	DLBA	?	„erstes LBA-Problem“
TM (= NTM)	DTM	ja	prinzipiell äquivalent

Beweisskizze: NTM durch DTM simulieren

Betrachte NTM N mit $\forall q, \gamma : |\Delta(q, \gamma)| \leq 2$.

$x \in L(N)$ gdw. eine *akzeptierende Berechnung* von $N(x)$ existiert, d.h. eine Folge von Konfigurationsübergängen

$$B_N^+ = k_0(x) \vdash_N \dots \vdash_N k_t = k_F,$$

wobei k_F eine Enkonfiguration ist. Eine DTM kann in $\leq 2^t$ Schritten alle möglichen Berechnungen von $N(x)$ der Länge t *simulieren* und findet auch B_N^+ . Ist jedoch $x \notin L(N)$, so terminiert $N(x)$ nicht und auch nicht die Simulation mit immer größeren t .

Bemerkung Formal muss noch genauer geklärt werden, *wie* die Simulation erfolgt: siehe dazu *universelle Turing-Maschine* (später).

Die wichtigsten Abschlusseigenschaften

4.6-6

	Schnitt	Vereinigung	Komplement	Produkt	Stern
REG	ja	ja	ja	ja	ja
DCFL	nein	nein	ja	nein	nein
CFL	nein	ja	nein	ja	ja
CSL	ja	ja	ja ¹⁰	ja	ja
RE	ja	ja	nein ¹¹	ja	ja

³vom r.A. zum DFA: Thompson-Konstruktion (1968) und zurück: Satz von Kleene (1956)

⁴Satz von Chomsky-Schützenberger (1963)

⁵Satz v. Istrail (1981)

4.6-7 Entscheidbarkeit

	Wortproblem	Leerheitsproblem	Äquivalenzproblem	Schnittproblem
REG	ja	ja	ja	ja
DCFL	ja	ja	ja	nein
CFL	ja	ja	nein	nein
CSL	ja	nein	nein	nein
RE	nein	nein	nein	nein

Laufzeitschranken für das Wortproblem

REG linear in der Wortlänge (siehe Paragraph 2.8-1)

DCFL linear (falls Sprache durch DPDA gegeben)

CFL kubisch (CYK-Algorithmus)

CSL exponentiell (NP-schwer)

4.6-8 Techniken/Werkzeuge

- kombinatorisch/logisch: z.B. Pumping-Lemmata, de Morgan, Anzahlargumente
- algorithmisch: z.B. Iteration bis zur Sättigung
- Reduktion: z.B. Äquivalenzproblem auf Leerheitsproblem
- Perspektivwechsel: z.B. Berechnung \leftrightarrow Ableitung
- Paarung/numerische Aufzählung von Strings und darauf aufbauende Algorithmen

5 Berechenbarkeit

5.1 Intuitive Berechenbarkeit

5.1-1 Was bedeutet „ $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ ist berechenbar“?

Intuitiv: Es gibt irgendein „Rechenverfahren“, das angewendet auf $x \in$ Definitionsbereich nach endlich vielen Schritten mit dem Ergebnis $f(x)$ stoppt.

Was ist mit „irgendein Rechenverfahren“ gemeint?

Dazu gehören sicher:

- DFA, PDA
- Ersetzungssysteme (wie Grammatiken, analog zu mathematischen Umformungsschritten)
- TM

⁶Satz von Immerman-Szelepcsenyi (1988) = Lösung des zweiten LBA-Problems

⁷Beweis später

- Python-Programme.

Aber: Was ist mit Parallelrechnern, Quantencomputern, DNA-Computern, ... zukünftigen Verfahren?

Beispiele (1): Einfache arithmetische Funktionen

5.1-2

Einige Beispiele um das Spektrum möglicher „Rechenverfahren“ wenigstens anzudeuten:

Nachfolger: gegeben $\text{bin}(n)$, gesucht $\text{bin}(n+1)$
(siehe Paragraph 4.2-7)

Addition: gegeben n, m , gesucht $n + m$ (jeweils binär)

- z.B. Schulmethode oder m mal den Nachfolger berechnen

Multiplikation: gegeben n, m , gesucht $n \cdot m$

- z.B. m mal: n aufaddieren (mit 0 beginnen), oder russische Bauernmultiplikation, oder Schulmethode, oder ...

Teilbarkeit durch 3 (oder irgendeine andere Zahl)

- z.B. früher betrachteter DFA oder geeignete Quersummenregel

Binomialkoeffizienten

- z.B. Definitionsformel anwenden oder Pascalsches Dreieck oder ...

Beispiele (2): Triviale(?) Funktionen

5.1-3

Konstante Funktionen

```
# c irgendein konstanter String
def func_c(x):
    return c
```

Die **nirgends definierte Funktion** Ω

$$\forall s : \Omega(s) = \infty,$$

wobei ∞ als Abkürzung für *nicht definiert* steht. Diese Funktion ist berechenbar!

Diese Abkürzung werden wir noch oft benutzen — bitte merken!

```
def Omega(x):
    while True:
        pass
    return '1'
# zur naechsten Iteration uebergehen
```

D_f bezeichnet den
Definitionsbereich von f

Bei partiellen $f : \{0, 1\}^* \rightrightarrows \{0, 1\}^*$ fordert man, dass das Verfahren für $x \in D_f$ mit $f(x)$ stoppt, aber für $x \notin D_f$ gar nicht terminiert— also insgesamt korrektes Verhalten auf *jeder* Eingabe.

5.1-4 Beispiele (3): Spaß mit π

Das Beispiel ist aus **[Schöning]**. Es geht es um Funktionen der Gestalt $\{0, 1, \dots, 9\}^* \rightarrow \{0, 1\}$ oder $\mathbb{N} \rightarrow \{0, 1\}$ (also „ja/nein-Entscheidungen“), die zusammenhängen mit d_π , der unendlich langen dezimalen Ziffernfolge von π .

π -Präfixe

$$\text{Pref}_\pi(x) = \begin{cases} 1 & \text{falls } x \text{ Präfix ist von } d_\pi, \\ 0 & \text{sonst.} \end{cases}$$

Beispiel: $\text{Pref}_\pi(3) = \text{Pref}_\pi(31) = \text{Pref}_\pi(314) \dots = 1$, aber z.B. $\text{Pref}_\pi(2) = 0$.

Die Funktion ist berechenbar durch numerische Verfahren beliebiger Genauigkeit.

π -Muster

$$\text{match}_\pi(x) = \begin{cases} 1 & \text{falls } x \text{ Teilstring ist von } d_\pi, \\ 0 & \text{sonst.} \end{cases}$$

Jede Ziffernfolge der Länge 5 kommt
„bereits“ innerhalb der ersten 10^8
Ziffern von π vor. Inzwischen sind
mindestens 10^{13} bekannt.

Es ist kein x mit $\text{match}_\pi(x) = 0$ bekannt. Es ist nicht bekannt, ob match_π berechenbar ist. Käme *jede* Ziffernfolge vor, dann wäre die Funktion konstant und damit berechenbar.

5.1-5 Beispiel (4): Mehr Spaß mit π

Die „Siebener-Funktion“ von π

$$\text{rep7}_\pi(n) = \begin{cases} 1 & \text{falls } \underbrace{77\dots7}_n \text{ Teilstring ist von } d_\pi, \\ 0 & \text{sonst.} \end{cases}$$

Diese Funktion ist berechenbar! Kommen beliebig lange 7er-Blöcke vor, so ist sie konstant 1. Anderenfalls gilt $\text{rep7}_\pi(n) = 1$, solange $n \leq n_0$ (für eine Maximallänge n_0), ansonsten ist der Funktionswert 0.

5.1-6 Beispiel (5): Ungelöste Probleme

Die „LBA-Funktion“

$$\text{lba}(n) = \begin{cases} 1 & , \text{ falls das LBA-Problem eine positive Lösung hat} \\ 0 & , \text{ sonst} \end{cases}$$

ist berechenbar, denn sie ist entweder konstant 0 oder konstant 1.

5.1-7 Es gibt Funktionen, die *nicht* intuitiv berechenbar sind

Mindestforderung

Ein „Rechenverfahren“ hat eine endliche Beschreibung $\in \Sigma^*$. Grundsätzlich kann es dann auch mit einer Bitfolge beschrieben werden (durch Kombination mit einer entsprechenden Bijektion $\Sigma^* \rightleftharpoons \{0, 1\}^*$).

Beweisskizze: Existenz intuitiv unberechenbarer Funktionen

1. Es gibt ebensoviele Bitfolgen $\in \{0, 1\}^*$ wie natürliche Zahlen (siehe Paragraph 4.4-2). Also ist die Menge der intuitiv berechenbaren Funktionen *abzählbar*.

2. Aber die Teilmenge $\mathcal{F}_{\{0,1\}^*} = \{f : \{0, 1\}^* \rightarrow \{0, 1\}\}$ ist nicht abzählbar!

Cantorsches **Diagonalargument**: Angenommen $\mathcal{F}_{\{0,1\}^*} = \{f_1, f_2, \dots\}$.

Dann ist jedoch die Funktion $f_D \in \mathcal{F}_{\{0,1\}^*}$ mit

$f_D(\mathbf{n}2\mathbf{b}(n)) = 1 - f_n(\mathbf{n}2\mathbf{b}(n))$ verschieden von allen f_n , im Widerspruch zur Annahme. \square

Eine Menge, die man „durchnummerieren“ kann, heißt *abzählbar*. Beispielsweise kann man alle Bitstrings mittels unserer Funktion $\mathbf{b}2\mathbf{n}$ nummerieren.

Bemerkung

Gleiches Argument: es gibt *überabzählbar* viele reelle Zahlen r . Also: es gibt intuitiv unberechenbare Funktionen der Form Pref_r .

Unzulänglichkeit des intuitiven Berechenbarkeitsbegriffs

5.1-8

Intuitive Berechenbarkeit genügt zum Nachweis

- der Berechenbarkeit einer konkreten Funktion/Klasse von Funktionen: (ein Verfahren klar beschreiben und sich von der Korrektheit überzeugen),
- oder der Existenz nicht-berechenbarer Funktionen.

Der Begriff ist aber zu unpräzise, um von einer *konkreten* Funktion nachzuweisen, dass *überhaupt kein* Verfahren zu ihrer Berechnung möglich ist!

Unser Plan

verschiedene formale Definitionen der Berechenbarkeit betrachten und ihre Äquivalenz durch gegenseitige Simulation nachweisen.

Als erstes werden wir *Turing-Berechenbarkeit* definieren und untersuchen.

5.2 Turing-Berechenbarkeit

On Computable Numbers, with an Application to the Entscheidungsproblem (A.M. Turing, 1936)

5.2-1

Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, *i.e.* on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we

The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite.

Videos

- <https://www.youtube.com/watch?v=FTSAiF9AHN4>
- <http://aturingmachine.com/>

5.2-2 Berechenbarkeit

Erinnerung: Konfigurationsübergänge

- $k \vdash^* k'$ bedeutet: es gibt eine Konfigurationsfolge

$$k \vdash k_1 \vdash \dots \vdash k_t \vdash k'$$

- Konfigurationen werden als $\alpha q \beta$ notiert (Inscript $\alpha \beta$, Kopf auf erstem Zeichen von β , aktueller Zustand q)

Definition

- **Wortfunktion** $f : \Sigma^* \rightrightarrows \Sigma^*$ heißt **Turing-berechenbar**, falls es eine DTM mit (o.B.d.A. einzigem) Endzustand q_F gibt, so dass für alle $x, y \in \Sigma^*$:

$$f(x) = y \text{ genau dann, wenn } q_0 x \vdash^* \square \dots \square q_F y \square \dots \square$$

- **Zahlfunktion** $f : \mathbb{N}^k \rightrightarrows \mathbb{N}$ heißt **Turing-berechenbar**, falls es eine DTM mit einzigem Endzustand q_F gibt, so dass für alle $n_1, \dots, n_k, m \in \mathbb{N}$:

$$f(n_1, n_2, \dots, n_k) = m \text{ gdw. } q_0 \text{bin}(n_1) \# \text{bin}(n_2) \# \dots \# \text{bin}(n_k) \vdash^* \square \dots \square q_F \text{bin}(m) \square \dots \square$$

Bemerkung

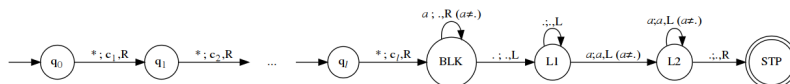
Beschränkung auf DTM ist keine Einschränkung: NTM durch DTM simulierbar

5.2-3 Beispiele Turing-berechenbarer Funktionen

- die Nachfolgerfunktion $s : \mathbb{N} \rightarrow \mathbb{N}$ mit $\forall n : s(n) = n + 1$ (siehe Paragraph 5.1-2)
- die nirgends definierte Funktion (zwei Implementationen):



- konstante Funktionen: $c(x) := c_1 c_2 \dots c_\ell \in \Sigma^*$



- Übungsaufgabe: Für jedes i ist die **Bitauswahl-Funktion** $\text{sel}_i : \{0, 1\}^* \rightarrow \{0, 1\}$ definiert durch

$$\text{sel}_i(x) = \begin{cases} x_i & \text{falls } |x| \geq i \\ \infty & \text{sonst.} \end{cases}$$

Turing-berechenbar

Berechnungen

5.2-4

Jede Konfiguration einer DTM hat höchstens eine Nachfolgekongfiguration.

Definition

Ist M eine DTM und $x \in \Sigma^*$, so bezeichne **comp_M(x)** die *eindeutig bestimmte Berechnung* von $M(x)$, d.h. die Konfigurationsfolge

$$k_0(x) \vdash_M k_1 \vdash_M k_2 \vdash_M \dots$$

Wir sagen „ M hält bei Start auf Eingabe x “ (kurz: **$M(x)$ terminiert**), falls $\text{comp}_M(x)$ eine Endkonfiguration enthält.

Ressourcen der Berechnung $\text{comp}_M(x) = k_0(x) \vdash k_1 \vdash k_2 \dots$

5.2-5

Von $M(x)$ beanspruchte **Rechenzeit**

= Anzahl der Übergänge in $\text{comp}_M(x)$ bis zur ersten Endkonfiguration:

$$T^M(x) = \begin{cases} \min\{i : k_i \text{ ist Endkonfiguration}\} & \text{falls } M(x) \text{ terminiert} \\ \infty & \text{sonst.} \end{cases}$$

Sei $t : \mathbb{N} \rightarrow \mathbb{N}$. M heißt **$t(n)$ -zeitbeschränkt**, falls für alle x : $T^M(x) \leq t(|x|)$.

Von $M(x)$ beanspruchter **Speicherplatz**

= Maximum der Konfigurationslängen von $k_i \in \text{comp}_M(x)$, wobei die **Länge** von $k = \alpha q \beta$ gegeben ist durch $|k| := |\alpha \beta|$:

$$S^M(x) = \max\{|k_i| : i \leq T^M(x)\}$$

Sei $s : \mathbb{N} \rightarrow \mathbb{N}$. M heißt **$s(n)$ -platzbeschränkt**, falls für alle x : $S^M(x) \leq s(|x|)$.

5.2-6 Rechenzeit und Speicherplatz sind „exponentiell verwandt“

Satz

Gelte $\forall n : t(n) \geq n$. Dann ist jede $t(n)$ -zeitbeschränkte DTM auch $t(n)$ -platzbeschränkt.

Beweis

In $t(n)$ Schritten werden höchstens $t(n)$ Bandzellen betreten. □

Satz

Sei M eine $s(n)$ -platzbeschränkte TM mit Zustandszahl $z = |Q|$ und Anzahl der Bandsymbole $b = |\Gamma|$. Dann ist M auch $c^{1+s(n)}$ -zeitbeschränkt für $c = z + b$.

Beweis

- es gibt nur $K_n = s(n) \cdot z \cdot b^{s(n)}$ Konfigurationen der Länge $s(n)$
- längere Berechnungen würden einen Zyklus enthalten und wären verkürzbar (ähnlich: Äbpumpen"beim Pumping-Lemma)
- es gilt $K_n \leq c^{1+s(n)} = (z + b)^{1+s(n)} = z^{1+s(n)} + (1 + s(n))zb^{s(n)} + \dots$ (Binomialentwicklung) □

5.2-7 Zusatzinformation: Hierarchie-Sätze (nicht klausurrelevant)

Für „zeitkonstruierbare“ Funktionen $f(n), g(n)$ gilt

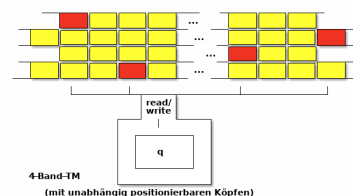
- $\text{DTIME}\left(o\left(\frac{f(n)}{\log f(n)}\right)\right) \not\subseteq \text{DTIME}(f(n))$
- $\text{NTIME}(f(n)) \not\subseteq \text{NTIME}(g(n))$ falls $f(n+1) = o(g(n))$

Für „raumkonstruierbare“ Funktionen $f(n)$ gilt

- $\text{xSPACE}(o(f(n))) \not\subseteq \text{xSPACE}(f(n))$, wobei xSPACE entweder für DSPACE oder NSPACE steht

5.2-8 Mehrband-Turing-Maschinen

unabhängig benutzbare Bänder für Zwischenergebnisse anstelle „Vor- und Zurückspulen“



Aktueller Zustand und k Fokus-Zeichen bestimmen die nächste Aktion:

- neuen Zustand annehmen,
- fokussierte Zellen neu beschreiben und

- Kopfbewegungen unabhängig für jedes Band ausführen.

Definition

Eine deterministische ***k*-Band-Turing-Maschine** (kurz: ***k*-DTM**) ist ein TM-Tupel $(Q, \Sigma, \Gamma, \delta, q_0, \square, F)$, mit Übergangsfunktion

$$\delta : Q \times \Gamma^k \rightrightarrows Q \times \Gamma^k \times \{L, R, S\}^k.$$

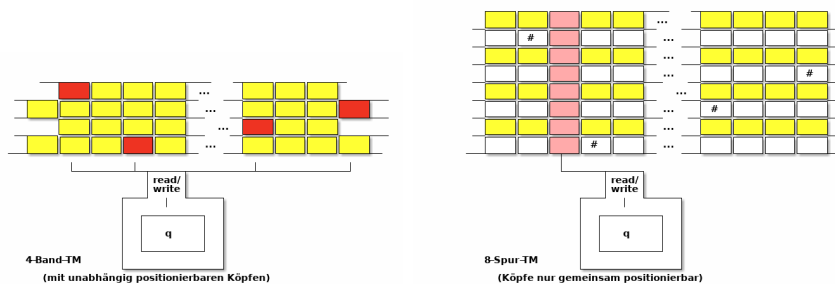
Mehrbandsimulation

5.2-9

Satz

Jede *k*-DTM *M* kann durch eine 1-DTM *M'* simuliert werden. Ist *M* *t*(*n*)-zeitbeschränkt mit *t*(*n*) ≥ *n*, so ist *M'* $\mathcal{O}(t^2(n))$ -zeitbeschränkt.

Beweisidee



Konfigurationen von *k*-DTM

5.2-10

Konfigurationen bei *k*-DTM beinhalten die vollständige Information über den Status einer Berechnung, also etwa so codiert

$$(\alpha_1 q \beta_1, \dots, \alpha_k q \beta_k),$$

wobei die $\alpha_i, \beta_i \in \Gamma^*$ die Bandinhalte sind (Köpfe jeweils auf erstem Zeichen des β -Anteils) und *q* der aktuelle Zustand ist. Konfigurationsübergänge sind entsprechend zu definieren.

Startkonfiguration

$$k_0(x) = (q_0 x, q_0 \square, \dots, q_0 \square),$$

d.h. die Eingabe wird auf dem ersten Band präsentiert, die anderen sind leer.

Die 2*k*-Spurmaschine *M'*

5.2-11

- 2*k*-Tupel als Symbole: Band ist in „Spuren“ eingeteilt: *k* „Datenspuren“ (mit den Bandinschriften der Bänder der *k*-DTM) und *k* „Positionsspuren“ (mit #-Marken für Kopfpositionen, sonst nur Blanks)

- Lesekopf = „Scanbalken“ über alle Spuren
- formale Anpassungen
 - $\Gamma' = \Gamma \cup (\Gamma \cup \{\#\})^{2k}$, $Q' = Q \cup \Gamma^k \times \{L, R, S\}^k$
 - $\delta' : Q' \times \Gamma' \rightarrow Q' \times \Gamma' \times \{L, R, S\}$

Start von $M'(x)$

- $x = x_1x_2...x_n$
- $k'_{-1}(x)$ entspricht Startkonfiguration $(q_0x, q_0\Box, ..., q_0\Box)$ von $M(x)$
- $k'_0 = q'_0(x_1, \#, \Box, \#, \Box, ..., \Box, \#)(x_2, \Box, \Box, ..., \Box)...(x_n, \Box, \Box, ..., \Box)$ = die „Spurendarstellung“ der Startkonfiguration mit Scanbalken ganz links

5.2-12 Simulation eines Schritts der k -DTM

Sei $k_0(x) \vdash k_1 \vdash ...$ die Berechnung von $M(x)$ und f.j. k_i bezeichne k'_i die entsprechende Konfiguration von $M'(x)$ (Spurendarstellung von k_i mit Scanbalken ganz links). Für jedes i wird Übergang $k_i \vdash_M k_{i+1}$ ersetzt durch Folge $k'_i \vdash_{M'}^* k'_{i+1}$ bestehend aus

1. Rechts-Scan über alle #: k Fokuszeichen/Bewegungen im Zustand speichern
2. Links-Scan bis ganz links: markierte Zeichen gemäß δ überschreiben, #-Marken aktualisieren (dabei $\leq k$ „Rücksetzer“ für Rechtsbewegungen)
3. Zustandswechsel gemäß δ

Zeitbedarf

- Initialisierung $k_0(x) = k'_{-1}(x) \vdash k'_0(x)$ kostet nur $2n$ Schritte.
- Simulation eines Schritts von M erfordert bis zu $2(s(n) + 2k)$ Schritte von M' (inkl. Rücksetzer), Platzschränke $s(n) \leq t(n)$ wegen $t(n) \geq n$
- also führt M' nur $O(t(n)^2)$ Schritte aus. □

5.2-13 Notation für Mehrbandmaschinen

- sei M eine 1-DTM, δ_M ihre partielle Übergangsfunktion
- für fixiertes k bezeichne $M(i)$ die k -DTM, die M auf Band i simuliert, andere Bänder unverändert lässt

Beispiel: $\delta_M(q, a) = (q', b, y)$ entspricht Band 2 und $k = 3$ der Festlegung

$$\forall c_1, c_3 \in \Gamma : \delta_{M(2)}(q, (c_1, a, c_3)) = (q', (c_1, b, c_3), (S, y, S))$$

Weitere Beispiele/Notationsvereinfachungen

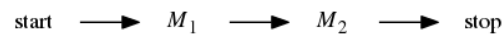
- „Band:=Band+1“ sei die früher betrachtete „Nachfolger-Maschine“

- entsprechend: „Band:=Band+1“(i) oder suggestiver „Band_i := Band_i + 1“
- ähnlich
 - „Band_i := Band_i – 1“
 - „Band_i := 0“ (Einzelzeichen)
 - „Band_i := Band_j“
- usw.

Strukturierte Programmierung: Komposition

5.2-14

DTMs $M_i = (Q_i, \Sigma, \Gamma_i, \delta_i, q_i, \square, F_i)$, $i = 1, 2$ sollen „hintereinandergeschaltet“ werden (programmiersprachlich: $M_1; M_2$):



Implementation

betrachte $M = (Q_1 \dot{\cup} Q_2, \Sigma, \Gamma_1 \cup \Gamma_2, \delta, q_1, \square, F_2)$ mit

$$\delta = \delta_1 \cup \delta_2 \cup \{\delta(q_F, a) = (q_2, a, S) : q_F \in F_1, a \in \Gamma_1\}$$

Beispiel: Addiere 3

„Band:=Band+1“, „Band:=Band+1“, „Band:=Band+1“

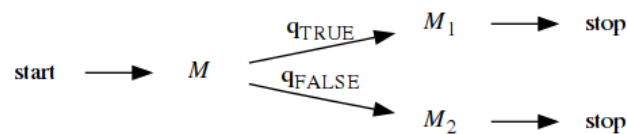
Strukturierte Programmierung: Bedingungen (Übungsaufgabe)

5.2-15

- sei M DTM mit Endzuständen q_{TRUE} und q_{FALSE} und M_1, M_2 DTM mit Startzuständen q_1, q_2
- ähnlich wie bei der Komposition kann man eine Maschine

IF M THEN M_1 ELSE M_2

mit diesem anschaulich beschriebenen Verhalten konstruieren:



Spezielle Tests

5.2-16

Die folgende Übergangsfunktion definiert eine Maschine „Band=0?“, die testet, ob sie auf der Bandinschrift 0 (Einzelzeichen) gestartet wurde:

- $\delta(q_0, a) = (q_{\text{FALSE}}, a, S)$ für $a \neq 0$
- $\delta(q_0, 0) = (q_1, 0, R)$

- $\delta(q_1, a) = (q_{\text{FALSE}}, a, S)$ für $a \neq \square$
- $\delta(q_1, \square) = (q_{\text{TRUE}}, \square, L)$

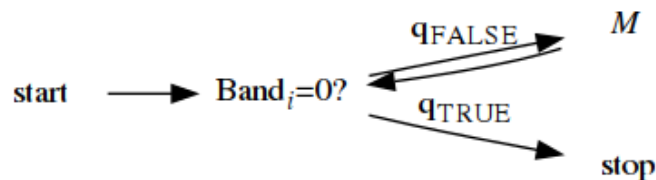
Die Band_i -Variante bezeichnen wir mit „ $\text{Band}_i=0?$ “.

5.2-17 Strukturierte Programmierung: Schleifen (Übungsaufgabe)

Ähnlich wie oben kann man eine Maschine

WHILE $\text{Band}_i \neq 0$ DO M

mit diesem Verhalten konstruieren:



5.2-18 Prozedurale Programmierung: Funktionsaufrufe

Maschine M_1 soll Maschine M_2 „aufrufen“ und deren Ergebnis weiterverarbeiten

Idee

Aufrufparameter/Rückgabewerte auf separaten Bändern übergeben, damit sich Berechnungen nicht gegenseitig stören

Implementation

ähnlich wie $M_1; M_2; M_1$, nur müssen für die Kontrollübergabe spezielle Zustände q_{call} und $q_{\text{return}} \in Q_1$ (Zustandsmenge von M_1) und entsprechende Übergänge vorgesehen werden.

- $Q = Q_1 \dot{\cup} Q_2$
- $\delta = \delta_1 \cup \delta_2 \cup \{\delta(q_{\text{call}}, a) = (q_2, a, S) : a \in \Gamma_1\} \cup \{\delta(q, a) = (q_{\text{return}}, a, S) : a \in \Gamma_2, q \in F_2\}$

Ist q_{return} erreicht, so beginnt die Weiterverarbeitung des Ergebnisses.

5.3 LOOP-, WHILE- und GOTO-Berechenbarkeit

5.3-1 Die Programmiersprache LOOP

Die Lexik

Variablen x_0, x_1, x_2, \dots , **Konstanten** $0, 1, 2, \dots$, **Symbole** $;$, $:=$, $+$, \div und **Schlüsselwörter** LOOP, DO, END

Syntax von LOOP-Programmen

- ist $c \in \mathbb{N}$ Konstante und sind x_i, x_j Variablen, so sind die folgenden Wörter LOOP-Programme:

$$x_i := x_j + c \text{ und } x_i := x_j \dot{-} c$$

- sind P_1 und P_2 LOOP-Programme, so auch

$$P_1; P_2$$

- ist x_i eine Variable und P ein LOOP-Programm, so auch

$$\text{LOOP } x_i \text{ DO } P \text{ END}$$

Weitere LOOP-Programme gibt es nicht.

Der Speichervektor

5.3-2

Beobachtung jedes LOOP-Programm enthält nur endlich viele Variablen (o.B.d.A. mit $x_0, x_1, \dots, x_k, x_{k+1}, \dots, x_m$ bezeichnet)

Definition

- der **Speicherzustand** (*Speichervektor*) $\mathbf{v} = (v_0, v_1, \dots, v_k, v_{k+1}, \dots, v_m)$ ist die Liste der aktuellen Werte vorkommender Variablen
- bei Eingabe $(n_1, n_2, \dots, n_k) \in \mathbb{N}^k$ ist der Startzustand $(0, n_1, \dots, n_k, 0, \dots, 0)$, d.h. x_1, \dots, x_k speichern die Eingabewerte, alle anderen den Wert 0

Übergangsfunktion

Wir definieren die Semantik von LOOP-Programmen durch ihre Wirkung auf den Speichervektor.

$$\delta(\cdot, P) : \mathbf{v}_{\text{vorher}} \mapsto \mathbf{v}_{\text{nachher}}$$

Semantik von LOOP-Programmen

5.3-3

Bezeichnung

$\mathbf{v}[x_i \leftarrow b]$ = Ergebnis der Ersetzung von v_i durch b in \mathbf{v}

Wirkung der Anweisungen

- **Zuweisung**

$$\delta(\mathbf{v}, x_i := x_j + c) = \mathbf{v}[x_i \leftarrow (v_j + c)]$$

$$\delta(\mathbf{v}, x_i := v_j \dot{-} c) = \mathbf{v}[x_i \leftarrow \max\{0, v_j - c\}]$$

$\dot{-}$ steht für die **modifizierte Subtraktion**: das Ergebnis ist stets ≥ 0 (siehe Paragraph 2.1-7)

Meyer, Albert R.; Ritchie, Dennis M. (1967). The complexity of loop programs. ACM '67: Proceedings of the 1967 22nd national conference. doi:10.1145/800196.806014

- **Komposition** $\delta(\mathbf{v}, P_1; P_2) = \delta(\delta(\mathbf{v}, P_1), P_2)$
- **LOOP-Anweisung** $\delta(\mathbf{v}, \text{LOOP } x_i \text{ DO } P \text{ END}) = \delta(\mathbf{v}, \underbrace{P; P; \dots; P}_{v_i})$

Die Anzahl der Iterationen steht vor Betreten der Schleife fest!!

5.3-4 LOOP-Berechenbarkeit

Definition

$f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt **LOOP-berechenbar**, falls ein LOOP-Programm P existiert, so dass für alle $n_1, n_2, \dots, n_k \in \mathbb{N}$ gilt

$$\delta((0, n_1, \dots, n_k, 0, \dots), P) = (f(n_1, \dots, n_k), \dots).$$

Programmstart: „Eingabevariablen“ x_1, \dots, x_k speichern die Eingabe

Programmende: **Akkumulatorvariable** x_0 speichert Funktionswert, die anderen Variablen speichern „irgend etwas“

5.3-5 Eigenschaften

Folgende Befehle können hinzugefügt werden, ohne die Menge berechenbarer Funktionen zu ändern:

- $x_i := x_j$ (wie $x_i := x_j + c$ mit $c = 0$)
- **IF** $x = 0$ **THEN** P **END** (mit offensichtlicher Semantik) kann simuliert werden durch

$y := 1; \text{LOOP } x \text{ DO } y := 0 \text{ END}; \text{LOOP } y \text{ DO } P \text{ END}$

5.3-6 Weitere Makros/Spracherweiterungen

- die **Addition** $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ ist LOOP-berechenbar:

$x_0 := x_1; \text{LOOP } x_2 \text{ DO } x_0 := x_0 + 1 \text{ END}$

und erlaubt somit die Hinzunahme von Befehlen wie $x_i := x_j + x_k$

- die **Multiplikation** $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ ist LOOP-berechenbar

$x_0 := 0; \text{LOOP } x_2 \text{ DO } x_0 := x_0 + x_1 \text{ END}$

beinhaltet zwei verschachtelte LOOP-Anweisungen

Übungsaufgabe

Geben Sie LOOP-Programme an, die folgende Funktionen berechnen

- **ganzzahlige Division** $(x, y) \mapsto \lfloor x/y \rfloor$ (Bezeichnung: $x \text{ DIV } y$)
- **Divisionsrest** $(x, y) \mapsto x \pmod{y}$ (Bezeichnung $x \text{ MOD } y$)

5.3-7 Von LOOP zu WHILE

Bei LOOP-Programmen gibt es keine Endlosschleifen.

Konsequenz

- LOOP-berechenbare Funktion sind total (siehe Paragraph 2.1-7)
- nicht alle Turing-berechenbaren Funktionen sind total (z.B. Ω)

\Rightarrow wir betrachten eine größere Klasse von Programmen: WHILE-Programme

Syntax von WHILE-Programmen

- jedes LOOP-Programm ist auch ein **WHILE-Programm**
- ist x_i Variable und P ein WHILE-Programm, dann auch

WHILE $x_i \neq 0$ DO P END

weitere WHILE-Programme gibt es nicht.

Offensichtlich beabsichtigte Semantik

„solange $x_i \neq 0$ wiederhole P “

Formale Definition der Semantik

5.3-8

die **Terminierungszahl** $\min T_i(\mathbf{v})$ ist definiert durch

$$\min\{k : \delta(\mathbf{v}, P^k) = (\dots, v_{i-1}, 0, v_{i+1}, \dots)\}$$

Definition (**Semantik der WHILE-Anweisung**)

$$\delta(\mathbf{v}, \text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}) = \begin{cases} \delta(\mathbf{v}, P^{\min T_i(\mathbf{v})}) & \text{falls } \min T_i(\mathbf{v}) < \infty \\ \infty & \text{sonst.} \end{cases}$$

D.h. der Speicherzustand einer nicht-terminierenden Schleife ist nicht definiert.
Zusätzlich: für beliebige WHILE-Programme Q sei

$$\delta(\infty, Q) = \infty$$

(undefinierter Speicherzustand bleibt undefiniert)

WHILE simuliert LOOP

5.3-9

LOOP x DO P END ist äquivalent zu

$y := x; \text{WHILE } y \neq 0 \text{ DO } y := y \div 1; P \text{ END}$

Definition

$f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt **WHILE-berechenbar**, falls ein WHILE-Programm P existiert, so dass für alle $n_1, n_2, \dots, n_k \in \mathbb{N}$ gilt

$$\delta((0, n_1, \dots, n_k, 0, \dots), P) = (f(n_1, \dots, n_k), \dots).$$

Insbesondere bedeutet das: Ist $f(n_1, \dots, n_k) = y$ und $y < \infty$, so stoppt $P(n_1, \dots, n_k)$ mit Wert y in x_0 . Falls $y = \infty$, so stoppt $P(n_1, \dots, n_k)$ gar nicht.

5.3-10 Turing-Maschinen simulieren WHILE-Programme

Satz

Jede WHILE-berechenbare Funktion ist auch TM-berechenbar.

Beweisskizze

- siehe Diskussion am Ende des Abschnitts „Turing-Berechenbarkeit“:
Zuweisung, Komposition und WHILE-Schleifen sind durch Mehrband-TM simulierbar
- i -tes Band der DTM entspricht Variable x_i
- Mehrband-DTM kann durch Einband-DTM simuliert werden. \square

Auch die Umkehrung gilt.

Zwischenschritt hierfür: GOTO-Programme

5.3-11 Syntax von GOTO-Programmen

- **GOTO-Programme** sind Folgen von markierten Anweisungen:

$$M_1 : A_1; M_2 : A_2; \dots; M_k : A_k$$

- **Marken** M_1, M_2, \dots sind unterscheidbar (wie Zeilennummern in Quelltext)
- **Anweisungen** A_i folgender Typen sind erlaubt
 - **Wertzuweisungen**: $x_i := x_j + c$ oder $x_i := x_j \div c$
 - **unbedingter Sprung**: **GOTO** M_j
 - **bedingter Sprung**: **IF** $x_i = c$ **THEN** **GOTO** M_j
 - **Stopanweisung**: **HALT**

für bel. Konstanten c

Vereinbarungen

- eine Marke kann weggelassen werden, wenn es im Programm dazu kein passendes GOTO gibt
- letzter Befehl ist entweder ein Sprung- oder ein Stoppbefehl

5.3-12 Semantik von GOTO-Programm $M_1 : A_1; \dots; M_k : A_k$

intuitiv: Programm beginnend bei $M_1 : A_1$ ausführen, dabei Sprünge folgen

Formal

- **Konfiguration** = Paar (\mathbf{v}, ℓ) aus Speichervektor und **Markenindex** (entspricht nächster auszuführender Anweisung)
- **Nachfolgekongfiguration** von (\mathbf{v}, ℓ) je nach Anweisungstyp von A_ℓ :
 - Wertzuweisung (wie bei LOOP-Programmen): $x_i := x_j \pm c$

$$(\mathbf{v}, \ell) \vdash (\delta(\mathbf{v}, A_\ell), \ell + 1)$$

- unbedingter Sprung: **GOTO** M_j : $(\mathbf{v}, \ell) \vdash (\mathbf{v}, j)$
- bedingter Sprung: **IF** $x_i = c$ **THEN** **GOTO** M_j

$$(\mathbf{v}, \ell) \vdash \begin{cases} (\mathbf{v}, j) & , \text{ falls } v_i = c \\ (\mathbf{v}, \ell + 1) & , \text{ sonst} \end{cases}$$

- Stopanweisung **HALT**

$$(\mathbf{v}, \ell) \vdash (\mathbf{v}, 0)$$

GOTO-Berechenbarkeit

5.3-13

- **Startkonfiguration** auf Eingabe (n_1, \dots, n_k) :

$$k_0(n_1, \dots, n_k) := ((0, n_1, n_2, \dots, n_k, 0, \dots), 1)$$

- **Endkonfiguration**:

$$(\mathbf{v}, 0) \text{ für irgendein } \mathbf{v}$$

Definition

$f : \mathbb{N}^k \rightarrow \mathbb{N}$ heißt **GOTO-berechenbar**, falls ein GOTO-Programm P existiert, so dass für alle natürlichen Zahlen n_1, n_2, \dots, n_k und m gilt:

- falls $f(n_1, \dots, n_k) = m$, so gilt $k_0(n_1, \dots, n_k) \vdash^* ((m, \dots), 0)$
- falls $f(n_1, \dots, n_k) = \infty$, so gilt $\forall m : k_0(n_1, \dots, n_k) \not\vdash^* ((m, \dots), 0)$

GOTO simuliert WHILE

5.3-14

Satz

Jede WHILE-berechenbare Funktion ist GOTO-berechenbar.

Beweis

$$\text{WHILE } x_i \neq 0 \text{ DO } P \text{ END}$$

ist gleichbedeutend mit

$$M_1 \text{ IF } x_i = 0 \text{ THEN GOTO } M_2; P; \text{GOTO } M_1;$$

M_2 HALT

5.3-15 WHILE simuliert GOTO

Satz

Jede GOTO-berechenbare Funktion ist auch WHILE-berechenbar. Es gilt sogar: Jedes GOTO-Programm kann durch ein WHILE-Programm simuliert werden mit nur einer einzigen WHILE-Schleife!

Beweisidee

- betrachte GOTO-Programm

$$M_1 : A_1; M_2 : A_2; \dots; M_k : A_k$$

- GOTO-Konfiguration hat Gestalt (\mathbf{v}, ℓ)
- Konfigurationsfolgen des GOTO-Programms übersetzen in Zustandsfolge des Speichervektors eines WHILE-Programms
- dazu führe zusätzliche Variable *count* ein, die den aktuellen Markenindex hält
- solange $count = i, i \neq 0$ führe die **kombinierte Aktion A'_i** aus, die angegeben wird durch:
 - die originale Anweisung A_i kombiniert mit
 - einer geeigneten Aktualisierung von *count* (Sprung oder „normal weiter“?)

5.3-16 Die kombinierte Aktion A'_i :

Zuweisungen ersetze $x_j := x_\ell \pm c$; durch $x_j := x_\ell \pm c$; $count := count + 1$;

unbedingter Sprung ersetze GOTO M_n durch $count := n$;

bedingter Sprung ersetze IF $x_j = c$ THEN GOTO M_n durch IF $x_j = c$ THEN $count := n$ ELSE $count := count + 1$ END

Stopp ersetze HALT durch $count := 0$

5.3-17 Beweis

- das GOTO-Programm ist äquivalent zu

$count := 1$; WHILE $count \neq 0$ DO

 IF $count = 1$ THEN A'_1 END

 IF $count = 2$ THEN A'_2 END

...

IF $count = k$ THEN A'_k END END

- **Beobachtung** IF-ELSE-Anweisungen können ohne WHILE simuliert werden □

Kleene-sche Normalform für WHILE-Programme

5.3-18

Folgerung

Jede WHILE-berechenbare Funktion kann durch ein WHILE-Programm mit nur einer WHILE-Schleife berechnet werden.

Beweis

WHILE-Programm P berechne f

- simuliere P durch GOTO-Programm P'
- simuliere P' durch WHILE-Programm P'' mit nur einer WHILE-Schleife

□

GOTO-Programme und Turing-Maschinen

5.3-19

Satz

Jede Turing-berechenbare Funktion ist auch GOTO-berechenbar.

Beweisidee

- sei M Turing-Maschine, die f berechnet
- codiere M -Konfigurationen geeignet durch Variablenwerte $x, y, z \in \mathbb{N}$ eines GOTO-Programms
- simuliere M 's Konfigurationsübergänge durch GOTO-Programm P
- P hat die Gestalt „Initialisiere — Iteriere — Terminiere:“

$$M_1 : P_1; M_2 : P_2; M_3 : P_3$$

- P_1 : codiere Startkonfiguration in x, y, z wie nachfolgend beschrieben
- P_2 : solange noch kein Endzustand erreicht, aktualisiere x, y, z gemäß aktuellem Konfigurationsübergang
- P_3 : extrahiere berechneten Wert aus x, y, z und speichere ihn in x_0

Codierung der Konfigurationen

5.3-20

- sei $M = (Q, \Sigma, \Gamma, \delta, q_1, \square, F)$ mit $Q = \{q_1, \dots, q_k\}$, $\Gamma = \{a_1, \dots, a_m\}$
- wähle Stellenwertbasis $b = m + 1$

- x bzw. y entsprechen Inschriftabschnitten links/rechts vom Kopf, z dem Zustand genauer: Konfiguration $a_{i_1} \dots a_{i_p} q_\ell a_{j_1} \dots a_{j_r}$ wird codiert durch

$$x = (i_1 i_2 \dots i_p)_b, y = (j_r j_{r-1} \dots j_1)_b, z = \ell$$

wobei $(\dots)_b$ für die b -adische Codierung einer Zahl steht

- **Wichtig:** bei y umgekehrte Ziffernreihenfolge!

Beispiel

$\Gamma = \{\square = a_1, a_2, \dots, a_9\}$, wähle $b = 10$

Konfiguration $\square \square a_2 a_5 q_2 a_7 a_3 \square$ entspricht der Variablenbelegung
 $x = 1125, y = 137, z = 2$ (Dezimalwerte)

5.3-21 Iteration: Simulation der Konfigurationsübergänge

M_2 $a := y \text{ MOD } b$; # extrahiere aktuelles Zeichen IF $(z = 1)$ AND $(a = 1)$
 THEN GOTO M_{11} ; IF $(z = 1)$ AND $(a = 2)$ THEN GOTO M_{12} ; ... IF $(z = k)$
 AND $(a = m)$ THEN GOTO M_{km} ;

M_{11} UPDATE(1,1); # berechne Codierung der Folgekonfiguration GOTO
 M_2 ;

M_{12} UPDATE(1,2); # .. GOTO M_2 ;

⋮

M_{km} UPDATE(k, m); # .. GOTO M_2 ;

5.3-22 UPDATE

Beobachtung

- Inschriften aufeinanderfolgender Konfigurationen unterscheiden sich nur wenig (nur in den Bandzellen unmittelbar beim Lese-/Schreibkopf)
- \Rightarrow ihre Codierungen unterscheiden sich nur in den Einerziffern

Erinnerung

$$x = (i_1 i_2 \dots i_p)_b = i_1 \cdot b^{p-1} + i_2 \cdot b^{p-2} + \dots + i_{p-1} \cdot b + i_p$$

$$y = (j_r j_{r-1} \dots j_1)_b = j_1 + j_2 \cdot b + \dots + j_{r-1} \cdot b^{r-2} + j_r \cdot b^{r-1}$$

Beispiel

Linksbewegung des Kopfes: $\delta(q_i, a_j) = (q_{i'}, a_{j'}, L) \Rightarrow$ aktualisiere x, y, z wie folgt:

$z := i'$; $y := y \text{ DIV } b$; $y := b \cdot y + j'$; $y := b \cdot y + (x \text{ MOD } b)$; $x := x \text{ DIV } b$; bei $z_i \in F$ keine Wertänderungen, sondern nur GOTO M_3

5.3-23 Initialisierung und Terminierung

Initialisierung $x = 0$ (eindeutig, da Ziffer 0 nicht vorkommt), $y =$ angepasste Stellenwertinterpretation der Eingabezeichen, $z = 0$

Terminierung Extraktion des Endergebnisses aus y durch rück-angepasste Stellenwertinterpretation der Arbeitsbandzeichen

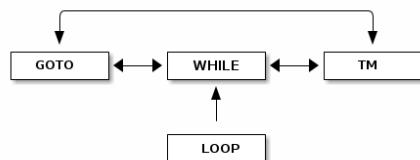
Anpassung der Stellenwerte

- nötig, da \square kein Eingabezeichen ist und mit Ziffernwert 1 interpretiert wird
- Beispiel: Bei dezimaler Eingabe ist $\# \Gamma \geq 11$ und $b \geq 12$ zu wählen.

Eingabe 42 führt dann zum Startwert $y = (4 + 1) + (2 + 1) \cdot 12$ \square

Bilanz

5.3-24



- Man sagt: WHILE und GOTO sind *Turing-vollständige Programmiersprachen*: sie können die gleichen Funktionen darstellen, die durch Turing-Maschinen berechenbar sind.
- LOOP ist nicht Turing-vollständig, denn alle LOOP-berechenbaren Funktionen sind total, aber es gibt auch partielle Turing-berechenbare Funktionen.

5.4 Primitiv und μ -rekursive Funktionen

Ein axiomatischer Zugang zur Berechenbarkeit

5.4-1

Induktive Idee

- betrachte gewisse *Basisfunktionen*, die als berechenbar angesehen werden
- konstruiere aus berechenbaren Funktionen weitere durch einfache Konstruktionschemata .

Basisfunktionen

- **konstante Funktionen**: $c : \mathbb{N}^0 \rightarrow \mathbb{N}$, für beliebiges $c \in \mathbb{N}$
- **Projektionen**: für $k \geq 1, 1 \leq i \leq k$ betrachte $\text{proj}_i^k : \mathbb{N}^k \rightarrow \mathbb{N}$ mit

$$\text{proj}_i^k(n_1, \dots, n_k) = n_i$$

Dieser (innermathematische) Ansatz ist übrigens die allererste Formalisierung des Berechenbarkeitsbegriffs. Ein lesenswerter Überblick über Geschichte und Ergebnisse der Rekursionstheorie ist in der Stanford Encyclopedia of Philosophy

- **Nachfolgerfunktion:** $\text{succ} : \mathbb{N} \rightarrow \mathbb{N}$ mit

$$\text{succ}(n) := n + 1$$

5.4-2 Konstruktionsschemata

Komposition

Seien $g_1, \dots, g_m : \mathbb{N}^k \rightarrow \mathbb{N}$ und $h : \mathbb{N}^m \rightarrow \mathbb{N}$ dann ist die Funktion $f := \kappa(h, (g_1, \dots, g_m)) : \mathbb{N}^k \rightarrow \mathbb{N}$ definiert durch $f(n_1, \dots, n_k) := h(g_1(n_1, \dots, n_k), \dots, g_m(n_1, \dots, n_k))$.

Primitive Rekursion (Skolem 1923)

Sei $g : \mathbb{N}^k \rightarrow \mathbb{N}$ Verankerungsfunktion und $h : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$ die Rekursionsvorschrift. Definiere $\rho(g, h) := f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ durch

$$\begin{aligned} f(x_1, \dots, x_k, 0) &:= g(x_1, \dots, x_k) \\ f(x_1, \dots, x_k, n+1) &:= h(x_1, \dots, x_k, n, f(x_1, \dots, x_k, n)). \end{aligned}$$

Bemerkung Aus dem Induktionsaxiom (Peano-Arithmetik) folgt, dass $\rho(g, h)$ wohldefiniert ist: jedem Tupel im Definitionsbereich wird genau ein Funktionswert zugeordnet.

5.4-3 Primitiv-rekursive Funktionen

Beispiel: Die Fakultätsfunktion ist eine 1-stellige Funktion

- **Verankerung** $f(0) := 1$ ist eine Konstante (d.h. 0-stellige Funktion)
- **Rekursionsvorschrift** $f(n+1) := (n+1) * f(n) = \text{Produkt von } (n+1) \text{ und } f(n)$ (Produkt ist eine 2-stellige Funktion)

Definition

Die Klasse PREK der **primitiv-rekursiven Funktionen** besteht aus den Basisfunktionen und allen Funktionen, die sich daraus konstruieren lassen durch endlich viele Anwendungen von Komposition und primitiver Rekursion.

5.4-4 Beispiele

- **einstellig konstante Funktion** (mit Wert 0)

$$\begin{aligned} c0(0) &= 0() \\ c0(n+1) &= \text{proj}_2^2(n, c0(n)) \end{aligned}$$

- **Addition**

$$\begin{aligned}\text{add}(x, 0) &= \text{proj}_1^1(x) \\ \text{add}(x, n+1) &= \text{succ}(\text{proj}_3^3(x, n, \text{add}(x, n)))\end{aligned}$$

- **Multiplikation**

$$\begin{aligned}\text{mult}(x, 0) &= c_0(x) \text{ (einstellig konstante Funktion mit Wert 0)} \\ \text{mult}(x, n+1) &= \text{add}(x, \text{proj}_3^3(x, n, \text{mult}(x, n)))\end{aligned}$$

- **Potenzfunktion** usw.

Erste Beobachtungen

5.4-5

- Primitiv-rekursive Funktionen sind *totale Funktionen* (auf allen Eingaben definiert)
- Vertauschung, Gleichsetzung von Argumenten: Beispiel:
 $g(a, b, c) = f(b, b, c, a)$ kann definiert werden durch Komposition

$$g(a, b, c) = f(\text{proj}_2^3(a, b, c), \text{proj}_2^3(a, b, c), \text{proj}_3^3(a, b, c), \text{proj}_1^3(a, b, c))$$

- **Folgerung** Primitive Rekursion *muss nicht* unbedingt in Bezug auf letztes Argument erfolgen.

Weitere Beispiele für primitiv rekursive Funktionen

5.4-6

- **modifizierte Vorgängerfunktion** $\text{pred}(n) = \max(n-1, 0)$

siehe Paragraph 2.1-7

$$\begin{aligned}\text{pred}(0) &= 0 \\ \text{pred}(n+1) &= n = \text{proj}_1^2(n, \text{pred}(n))\end{aligned}$$

- **modifizierte Subtraktion** $\text{sub}(x, y) = \max(x-y, 0)$

$$\begin{aligned}\text{sub}(x, 0) &= x \\ \text{sub}(x, n+1) &= \text{pred}(\text{sub}(x, n))\end{aligned}$$

- **Binomialkoeffizient über 2** $\binom{n}{2}$

$$\begin{aligned}\binom{0}{2} &= 0 \\ \binom{n+1}{2} &= \binom{n}{2} + n\end{aligned}$$

- Übungsaufgabe Binomialkoeffizienten $\binom{n}{k}$ für fixes k

Erinnerung: Cantorsche Paarungsfunktion

5.4-7

siehe Paragraph 4.4-3

- Codierung von Zahlpaaren durch einzelne Zahlen:

$x \backslash y$	0	1	2	3	...
0	0	2	5	9	...
1	1	4	8	13	...
2	3	7	12	18	...
3	6	11	17	24	...
...

- die Cantorsche Paarungsfunktion

$$\pi(x, y) = \binom{x + y + 1}{2} + y$$

ist primitiv rekursiv

- die Verallgemeinerung auf k -Tupel ist auch primitiv-rekursiv:

$$\langle n_0, \dots, n_k \rangle = \pi(n_0, \pi(n_1, \dots, \pi(n_k, 0) \dots))$$

5.4-8 Umkehrung: Die Cantorprojektionen

- Cantor-Projektionen: $n \mapsto (\pi_L(n), \pi_R(n))$, d.h. $\pi(\pi_L(n), \pi_R(n)) = n$
- Verallgemeinerung auf Tupel $n := \langle n_0, n_1, \dots, n_k \rangle$ (siehe frühere Betrachtung(Paragraph 4.4-5)): sind π_L, π_R primitiv-rekursiv, dann auch die Komponentenfunktionen

Berechnung der Cantor-Projektionen

Beispiel: $\pi_R(n) = n - t_{max}$, wobei

$$t_{max} = \max\{t \in \{0, 1, 3, 6, 10, \dots, \binom{x}{2}, \dots\} : t \leq n\}.$$

Kann man diese Funktion primitiv-rekursiv definieren?

5.4-9 Hilfsmittel: Primitiv-rekursive Prädikate

Ein **zahlentheoretisches Prädikat** ist eine Funktion, die einer Zahl einen Wahrheitswert $\in \{0, 1\}$ zuordnet (Beispiel: Primzahleigenschaft).

- **beschränkter max-Operator** ordnet einem Prädikat eine Zahlenfunktion zu

$$q(n) := \begin{cases} \max\{x \leq n \mid P(x)\} & \text{falls Menge nicht leer} \\ 0 & \text{sonst.} \end{cases}$$

- **beschränkter Existenzquantor** ordnet einem Prädikat ein neues zu:

$$Q(n) \equiv \exists x \leq n : P(x)$$

Satz

Ist $P : \mathbb{N} \rightarrow \{0, 1\}$ primitiv rekursiv, dann auch $q = \text{max } P$ und $Q = \exists_{\leq} P$.

$$\begin{aligned}
q(0) &= 0 \\
q(n+1) &= \begin{cases} n+1 & , \text{ falls } P(n+1) \\ q(n) & , \text{ sonst} \end{cases} \\
&= q(n) + P(n+1) \cdot (n+1 - q(n))
\end{aligned}$$

$$\begin{aligned}
Q(0) &= P(0) \\
Q(n+1) &= P(n+1) + Q(n) - P(n+1) \cdot Q(n)
\end{aligned}$$

□

Berechnung mittels $\max P$ und $\exists \leq P$

5.4-11

Sei $\pi(x, y) = \binom{x+y+1}{2} + y$ und seien $\pi_L, \pi_R : \mathbb{N} \rightarrow \mathbb{N}$ definiert durch $\pi(\pi_L(n), \pi_R(n)) = n$.

Behauptung

π_L, π_R sind primitiv-rekursiv.

Beweis

π ist bijektiv und $\pi_L(n) \leq n, \pi_R(n) \leq n$, d.h. es genügt vollständige Suche in einem beschränkten Bereich:

$$\begin{aligned}
z(\ell, m, n) &= \max\{z \leq \ell \mid \exists s \leq m : \pi(z, s) = n\} \\
\pi_L(n) &= z(n, n, n)
\end{aligned}$$

$$\begin{aligned}
s(\ell, m, n) &= \max\{s \leq \ell \mid \exists z \leq m : \pi(z, s) = n\} \\
\pi_R(n) &= s(n, n, n)
\end{aligned}$$

Primitive Rekursion und LOOP-Berechenbarkeit

5.4-12

Satz

Die Klasse der primitiv-rekursiven Funktionen stimmt genau überein mit der Klasse der LOOP-berechenbaren Funktionen.

Beweis (Teil 1): Jede primitiv-rekursive Funktion ist LOOP-berechenbar

strukturelle Induktion über Aufbau von f

- Basisfunktionen (Konstante, Projektionen, Nachfolger): offenbar LOOP-berechenbar
- $f = h \circ (g_1, \dots, g_k)$ (Komposition primitiv rekursiver Funktionen)

- nach Ind.vor. \exists LOOP-Programme für h, g_1, \dots, g_k
- geeignete Hintereinanderausführung/Zwischenspeicherung berechnet f
- sei f definiert durch primitive Rekursion aus LOOP-berechenbarer Verankerung g und Rekursionsvorschrift h , also:

$$\begin{aligned} f(0, x_1, \dots, x_r) &= g(x_1, \dots, x_r) \\ f(n+1, x_1, \dots, x_r) &= h(x_1, \dots, x_r, n, f(n, x_1, \dots, x_r)) \end{aligned}$$

folgendes LOOP-Programm berechnet f :

$y := g(x_1, \dots, x_r); k := 0; \text{LOOP } n \text{ DO } y := h(x_1, \dots, x_r, k, y); k = k + 1 \text{ END}$

5.4-13 Beweis Teil 2 (Anfang)

- werde $f : \mathbb{N}^r \rightarrow \mathbb{N}$ durch das LOOP-Programm P berechnet mit Variablen x_0, x_1, \dots, x_k ($k \geq r$)
- zeigen zunächst per Induktion über Aufbau von P : \exists primitiv-rekursive Funktion $f_P : \mathbb{N} \rightarrow \mathbb{N}$, die „ P ’s Wirkung beschreibt“, d.h.
sind die Variablen mit a_0, \dots, a_k initialisiert und b_0, \dots, b_k die Endwerte nach Ausführung von P , so gilt

$$f_P(\langle a_0, \dots, a_k \rangle) = \langle b_0, \dots, b_k \rangle$$

- **Zuweisung** $P \ x_i := x_j \pm c$ wird beschrieben durch

$$f_P(n) := \langle n_0(n), \dots, n_{i-1}(n), n_j(n) \pm c, n_{i+1}(n), \dots, n_k(n) \rangle$$

- **Komposition** $P \ Q; R$ wird beschrieben durch

$$f_P(n) := g_R(g_Q(n))$$

5.4-14 Beweis Teil 2 (Ende)

- **LOOP-Anweisung** $P \ \text{LOOP } x_i \ \text{DO } Q \ \text{END}$
 - nach Induktionsvoraussetzung gibt es primitiv-rekursive Funktion $f_Q : \mathbb{N} \rightarrow \mathbb{N}$, die Q ’s Wirkung beschreibt
 - definiere nun mit primitiver Rekursion

$$\begin{aligned} h(0, x) &= x \\ h(n+1, x) &= f_Q(h(n, x)) \end{aligned}$$

- Beobachtung: $h(n, x) =$ Wert von $x = \langle x_0, \dots, x_k \rangle$ nach n Iterationen von Q
- also wird P ’s Wirkung beschrieben durch

$$f_P(n) = h(n_i(n), x)$$

- **Finale** f_P primitiv-rekursiv $\Rightarrow f$ primitiv-rekursiv, denn $f(n_1, \dots, n_r) =$ Wert von x_0 nach Ausführung von P , also:

$$f(n_1, \dots, n_r) = n_0(f_P(\langle 0, n_1, \dots, n_r, \underbrace{0, \dots, 0}_{k-r} \rangle))$$

□

μ -rekursive Funktionen

5.4-15

- sei $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ (ggf. partielle) Funktion
- der μ -Operator definiert eine neue (partielle) Funktion $g = \mu f : \mathbb{N}^k \rightarrow \mathbb{N}$:

$$g(x_1, \dots, x_k) = \min \{n \mid f(n, x_1, \dots, x_k) = 0 \wedge \forall m < n : f(m, x_1, \dots, x_k) \neq 0\},$$

mit $\min \emptyset = \infty$

- **Übungsaufgabe** Seien $f, g : \mathbb{N}^2 \rightarrow \mathbb{N}$ gegeben durch $f(x, y) = y \dot{-} x^2$ und $g(x, y) = y \dot{-} 2^x$.

Geben Sie $\mu f(y)$ und $\mu g(y)$ an.

Definition

Die Klasse REK der μ -rekursiven Funktionen besteht aus den Basisfunktionen und allen Funktionen, die sich daraus konstruieren lassen durch endlich viele Anwendungen von Komposition, primitiver Rekursion und μ -Rekursion.

μ -Rekursion und WHILE-Berechenbarkeit

5.4-16

Satz

Die Klasse der μ -rekursiven Funktionen stimmt genau überein mit der Klasse der WHILE-berechenbaren Funktionen.

Beweis

erweitere vorigen Beweis um Behandlung von WHILE bzw. μ

- (1) falls $g = \mu f$, wobei f WHILE-berechenbar, so berechne g durch:

$$\begin{aligned} x_0 &:= 0; y := f(0, x_1, \dots, x_k); \text{ WHILE } y \neq 0 \text{ DO} \\ x_0 &:= x_0 + 1; y := f(x_0, x_1, \dots, x_k) \text{ END} \end{aligned}$$

- (2) sei $P \equiv \text{WHILE } x_i \neq 0 \text{ DO } Q \text{ END}$ für WHILE-Programm Q

- definiere h wie oben, so dass $h(n, x) =$ Wert von $x = \langle x_0, \dots, x_k \rangle$ nach n Iterationen von Q
- entsprechend der Semantik von WHILE-Programmen:

$$f_P(x) = h(\mu(n_i \circ h)(x), x)$$

□

5.4-17 Kleene'scher Normalformsatz

Satz

Für jede n -stellige μ -rekursive Funktion f gibt es zwei $(n+1)$ -stellige primitiv-rekursive Funktionen p und q , so dass f darstellbar ist als

$$f(x_1, \dots, x_n) = p(x_1, \dots, x_n, \mu q(x_1, \dots, x_n)).$$

Kleene, Stephen C. (1936).
„ λ -definability and recursiveness“.
Duke Mathematical Journal. 2 (2):
340–352.

Beweis

- μ -rekursive Funktion \rightsquigarrow WHILE-Programm
- WHILE-Programm \rightsquigarrow GOTO-Programm
- GOTO-Programm \rightsquigarrow WHILE-Programm mit nur einer WHILE-Anweisung
- WHILE-Programm mit nur einer WHILE-Anweisung \rightsquigarrow μ -rekursive Funktion mit nur einer Anwendung des μ -Operators □

5.5 Die Churchsche These

5.5-1 Ein informaler Blick auf den λ -Kalkül

siehe Paragraph 4.4-1

Erinnerung λ -Ausdrücke beschreiben Zuordnungen „an sich“ als Objekte

```
lambda x: x+2      # "anonyme" Zuordnung: dem x wird x+2 zugeordnet
f = lambda x: x+2  # die Zuordnung unter dem Namen f speichern
```

Funktionale: Funktionen, die auf Funktionen operieren

```
g = lambda f: lambda x: f(f(x)) # f |--> f(f(?)) Funktion zweiter Ordnung
h = g(lambda x: x*x)           # Quadrat(?) |--> Quadrat(Quadrat(?))
h(2) == 16                      # (2^2)^2
# auch Funktionen dritter, vierter, ... Ordnung
```

siehe Paragraph 5.4-2

Beispiel: Die Operatoren κ (Komposition), ρ (primitive Rekursion) und μ sind Funktionen zweiter Ordnung. Der λ -Kalkül hat entfernte Ähnlichkeit mit dem Stufenaufbau der Mengenlehre und kann als formale Grundlage der Mathematik und für Programmiersprachen dienen (funktionales Paradigma: LISP, Haskell, ...).

siehe Paragraph 1.5-1

Satz (Church 1936, Kleene 1936)

Die Klasse der λ -definierbaren Funktionen ist identisch mit REK.

5.5-2 Bilanz

Wir haben sehr verschiedene Modelle für die intuitive Berechenbarkeit kennengelernt ...

- Turing-Maschinen

- GOTO-Programme (Assembler)
- WHILE-Programme (Iteration)
- μ -rekursive Funktionen (Rekursion)
- λ -definierbare Funktionen (funktionale Programmierung)

und (z.T.) formal nachgewiesen, dass sie alle die gleiche Klasse von Funktionen beschreiben: jedes der genannten Modelle kann die anderen simulieren.

Dies gilt auch für weitere Modelle: Markov-Algorithmen, Random Access Maschinen, Quantencomputer, DNA-Computer, ...

Die **Churchsche These**

5.5-3

Diese Erkenntnisse haben zu folgender Überzeugung geführt:

Church-Turing-These

Die Klasse der *Turing-berechenbaren* Funktionen stimmt genau mit der Klasse der im intuitiven Sinne berechenbaren Funktionen überein.

Status

Die These ist nicht formal beweisbar, da jede (zum Beweis ja notwendige) Formalisierung potentiell eine Einschränkung der intuitiven Berechenbarkeit darstellt.

Sie ist höchstens widerlegbar und zwar durch Angabe eines intuitiven Verfahrens, das Turing-Maschinen beweisbar *nicht* simulieren können.

5.6 Die Ackermann-Funktion

Motivation

5.6-1

Erinnerung: PREK = primitiv-rekursive = LOOP-berechenbare Funktionen

PREK ist sehr reichhaltig

Jede totale Funktion, die mit Zeitschranke $f(n) \in \text{PREK}$ berechnet werden kann, ist LOOP-berechenbar.

Beispiel: $n^2, \lceil \log n \rceil n^2, 2^n, 2^{2^n}, \dots \Rightarrow$ „gefühlte“ jede praktisch-relevante Funktion $\in \text{PREK}$

Beweisidee

ersetze WHILE $x \neq 0$ DO P END durch

$$y = f(|x|); \text{ LOOP } y \text{ DO IF } x \neq 0 \text{ THEN P END}$$

ABER $\text{REK} \cap \text{TOTAL} \not\subseteq \text{PREK}$

Beweisidee Diagonalisierung

- PREK ist rekursiv aufzählbar: $w \in \Sigma^*$ aufzählen und PREK-Definitions-Syntax prüfen \Rightarrow Funktionen $\{f_1, f_2, f_3, \dots\} = \text{PREK}$

- \Rightarrow Diagonalfunktion $g(n) := f_n(n) + 1$ ist in $\text{TOTAL} \setminus \text{PREK}$ Ist g berechenbar? Ja! (folgt aus Existenz der universellen Turing-Maschine, s. Paragraph 5.7-7)

5.6-2 Berechenbare totale *nicht* primitiv-rekursive Funktionen

Vermutung & Problem (Hilbert 1925)

Primitive Rekursion ist unzureichend: Man gebe eine totale, durch endliche Regelwerke definierte Funktion an, die nicht primitiv-rekursiv ist.

in Anlehnung an: Hilbert, D. Über das Unendliche. Math. Ann. 95, 161–190 (1926)

Beispiel: Die Ackermann-Funktion $a : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$

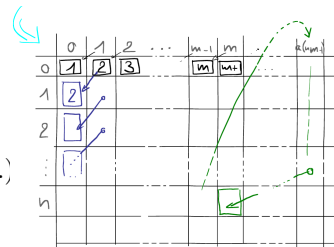
Ackermann, W. Zum Hilbertschen Aufbau der reellen Zahlen. Math. Ann. 99

$$\begin{aligned} a(0, m) &= m + 1 \text{ für } m \geq 0 \\ a(n, 0) &= a(n - 1, 1) \text{ für } n \geq 1 \\ a(n, m) &= a(n - 1, a(n, m - 1)) \text{ für } n, m \geq 1. \end{aligned}$$

Die ursprüngliche Definition lautete etwas anders

letzte Gleichung umgeschrieben:

$$a(n, m) = \underbrace{a(n - 1, a(n - 1, \dots, a(n - 1, 1) \dots))}_{(m+1) \text{ mal}}$$



5.6-3 Die Ackermann-Funktion ist intuitiv berechenbar

```
def ack(n,m):
    if n == 0:
        return m+1
    if m == 0:
        return ack(n-1,1)
    return ack(n-1,ack(n,m-1))
```

Wir diskutieren später ein WHILE-Programm für die Ackermann-Funktion \Rightarrow die Funktion ist auch im formalen Sinn berechenbar.

Übungsaufgabe: Einige Wachstums-Kostproben

Sei $A_n(m) := a(n, m)$ eine „Ackermann-Zeile“. Vollständige Induktion zeigt:

- $A_1(m) = m + 2$
- $A_2(m) = 2m + 3$
- $A_3(m) = 2^{m+3} - 3$
- $A_4(m) = \underbrace{2^{2^{\dots^2}}}_{(m+3) \text{ Zweien}} - 3$

- sei P ein LOOP-Programm und x_0, \dots, x_k die darin vorkommenden Variablen
- die **Wachstumsfunktion** $w_P : \mathbb{N} \rightarrow \mathbb{N}$ ist wie folgt definiert:
 - **Bezeichnung** $n_0, \dots, n_k = \text{Startwerte der } x_i \Rightarrow n'_0, \dots, n'_k \text{ zugehörige Endwerte nach Abarbeitung von } P$
 - definiere

$$w_P(n) = \max \left\{ \sum_{i \geq 0} n'_i \mid \sum_{i \geq 0} n_i \leq n \right\}$$

- **Beobachtung** $w_P(n)$ ist eine obere Schranke für die „jemals“ von P auf Eingaben $\leq n$ produzierbaren Werte

Lemma („Ackermann-Benchmark“)

Für jedes LOOP-Programm P gibt es eine Konstante n_P , so dass für alle m :

$$w_P(m) < a(n_P, m).$$

□

„Ackermann ist zu schnell für LOOP“

5.6-5

Satz

Die Ackermann-Funktion ist nicht LOOP-berechenbar.

Beweis

- wäre a LOOP-berechenbar, so auch $g(n) = a(n, n)$
- sei P LOOP-Programm für g und n_P Konstante, so dass für alle m gilt:

$$w_P(m) < a(n_P, m)$$

- für $n = n_P$ ergibt sich ein Widerspruch:

$$g(n_P) \leq w_P(n_P) < a(n_P, n_P) = g(n_P)$$

- folglich ist a *nicht* LOOP-berechenbar

WHILE-Programm für die Ackermann-Funktion

5.6-6

Laufzeitstapel: Jede rekursive Funktion kann unter Verwendung eines Keller-Speichers (Stack) rein iterativ programmiert werden.

Übungsaufgabe

Geben Sie ein WHILE-Programm zur Berechnung der Ackermann-Funktion an.

Nutzen Sie dazu neben WHILE-DO-END und IF-THEN-END etc. folgende Befehle:

INIT(t) initialisiert Variable t als leeren Stack

PUSH(x, t) legt Wert x auf dem Stack t ab

$y := \text{POP}(t)$ entfernt obersten Wert vom Stack t und speichert ihn in y

SIZE(t) $\neq 1$ testet, ob Stack t nichtleer ist („nur Kellerstart-Symbol enthält“)

5.6-7 Stack-Implementierung mit Cantor-Paarung/-Projektionen

Erinnerung $\pi(\pi_L(n), \pi_R(n)) = n$

Idee Stackinhalt $(n_1, n_2, \dots, n_k]$ durch $\pi(n_1, \pi(n_2, \dots, \pi(n_k, 0) \dots))$ codieren

- Implementierung der Stack-Operationen:

INIT(t) entspricht $t := 0$

PUSH(x, t) entspricht $t := \pi(x, t)$

$y := \text{POP}(t)$ entspricht $y := \pi_L(t); t := \pi_R(t)$

SIZE(t) $\neq 1$ entspricht $\pi_R(t) \neq 0$

Satz

Die Ackermann-Funktion ist total und berechenbar. □

5.7 Entscheidbarkeit und Semi-Entscheidbarkeit

5.7-1 Definition

Sei Σ ein Alphabet und $L \subseteq \Sigma^*$. Wir definieren

die **charakteristische Funktion** $\chi_L : \Sigma^* \rightarrow \{0, 1\}$ von L durch und die **partielle charakteristische Funktion** $\chi'_L : \Sigma^* \rightarrow \{0, 1\}$ durch

$$\chi_L(x) = \begin{cases} 1 & \text{falls } x \in L \\ 0 & \text{sonst} \end{cases} \quad \chi'_L(x) = \begin{cases} 1 & \text{falls } x \in L \\ \infty & \text{sonst.} \end{cases}$$

Die Sprache $L \subseteq \Sigma^*$ heißt

- **entscheidbar**, falls χ_L berechenbar
- **semi-entscheidbar**, falls χ'_L berechenbar ist (offenbar: semi-entscheidbar \iff wird durch eine TM erkannt).

Bemerkung

Wir setzen im folgenden Berechenbarkeit gleich mit „durch eine Python-Funktion berechenbar“.

5.7-2 Erste Beobachtungen

- ist L entscheidbar, dann auch semi-entscheidbar

type hints sind nicht verbindlich in Python, aber sie können die Lesbarkeit verbessern

```
def semidecideL(x: str) -> bool: # "Type hint"
    if decideL(x): return True # benutze Entscheider fuer L
    else:
        while (True):
            pass
```

- sind L und $K := \overline{L}$ (Komplement) semi-entscheidbar, dann auch entscheidbar

Beweis:

- modifiziere so, dass bei Laufzeit $> t$ Ergebnis `False` geliefert wird:
`semidecideL(x: str, t: int), semidecideK(x: str, t: int)`
- bei genügend großem t MUSS einer der Aufrufe `True` liefern:

```
def decideL(x):
    t = 0
    finishedL = finishedK = False
    while not finishedL and not finishedK:
        t += 1
        (finishedL, finishedK) = (semidecideL(x, t), semidecideK(x, t))
    return finishedL
```

Semi-Entscheidbarkeit und rekursive Aufzählbarkeit

5.7-3

Erinnerung: $L \subseteq \Sigma^*$ ist *rekursiv aufzählbar*, falls entweder $L = \emptyset$ oder es gibt eine Python-Funktion `enumerateL(n: int) -> str`, mit Ergebnismenge L

Satz

Die Sprache $L \subseteq \Sigma^*$ ist semi-entscheidbar gdw. sie rekursiv aufzählbar ist.

Beweis

semi-entscheidbar \Rightarrow rekursiv aufzählbar

siehe Beweis von $\mathcal{L}_0 \subseteq \text{RE}$ in Paragraph 4.5-4

rekursiv aufzählbar \Rightarrow semi-entscheidbar

```
def semidecideL(x):
    n = 0
    while enumerateL(n) != x:
        n += 1
    return True
```

□

Zusammenfassende Charakterisierung des Chomsky-Typs 0

5.7-4

Satz

Sei $L \subseteq \Sigma^*$ eine formale Sprache. Dann sind folgende Aussagen äquivalent:

- L wird durch eine (deterministische) Turing-Maschine erkannt,
- L ist semi-entscheidbar,

der Wertebereich einer
Aufzählungsfunktion für L ist genau
der Definitionsbereich der partiellen
charakteristischen Funktion von L
ist genau L
Äquivalenz der ersten beiden
Aussagen wurde in Paragraph 4.5-3
angekündigt und ist jetzt vollständig
bewiesen

- L ist rekursiv aufzählbar,
- L ist vom Typ 0,
- L ist Wertebereich einer berechenbaren Funktion.
- L ist Definitionsbereich einer berechenbaren Funktion,
- χ'_L ist μ -rekursiv (oder Turing-, WHILE-, GOTO-berechenbar, ...),

□

5.7-5 Das spezielle Halteproblem

Sei $M \mapsto \text{code}(M)$ eine *geeignete Codierung* (siehe nachfolgend) für Turing-Maschinen und bezeichne M_w die Turing-Maschine mit Code $w \in \{0, 1\}^*$. Das **spezielle Halteproblem** ist die Sprache

Terminierung: siehe Paragraph 5.2-4

$$K = \{w \in \{0, 1\}^* : M_w(w) \text{ terminiert}\}.$$

Geeignete Codierungen für Turing-Maschinen: *Gödelisierung*

(1) *irgendeine* Spezifikationsprache für Turing-Maschinen benutzen

siehe Paragraph 4.2-4

Beispiel: TM $(Q, \Sigma, \Gamma, \Delta, q_0, \square, F) \mapsto$ Python-Dictionary \mapsto Unicode-String

```
# Beispiel: Maschine M_Omega
{'Q': {'q0'}, 'Sigma': {'0', '1'}, 'Gamma': {'.', '0', '1'}, 'B': '.', 'q0': 'q0', 'F': set(), 'Delta': {}}
```

(2) Stringdarstellung binär umcodieren (Bsp. Unicode-String \mapsto Codepoint-Folge)

(3) Decodierung „mit JOKER“: Ist $w \in \{0, 1\}^*$ kein sinnvoller TM-Code, so bezeichne M_w irgendeine vorher fixierte Turing-Maschine (z.B. obige M_Omega).

Bemerkung

Für theoretische Diskussionen (z.B. Existenz universeller TM u.ä.) sind andere Codierungen bequemer, aber geeignet ist so gut wie jede. Insbesondere können wir die Arbeit einer Turing-Maschine auch durch Python-Quellcode beschreiben.

5.7-6 Die Unentscheidbarkeit des speziellen Halteproblems

Satz (Turing 1937)

Die Sprache K ist nicht entscheidbar.

Beweis

Aufgrund der Äquivalenz der Berechenbarkeitsbegriffe genügt es, die Python-Variante des speziellen Halteproblems zu betrachten:

$$K = \{w : w \text{ ist Python-Funktion mit String-Argument, und } w(w) \text{ terminiert}\}$$

- Annahme: Es gibt eine Implementierung `def chi_K(w): ...` von χ_K .
- Wir implementieren damit die neue Funktion

```
def seltsam(w):
    if chi_K(w) == 1:
        while True // Endlosschleife
```

- `seltsam(seltsam)` terminiert gdw. `seltsam(seltsam)` nicht terminiert!
- Folglich ist die Annahme falsch, K somit unentscheidbar.

□

Diskussion

5.7-7

Satz

K ist semi-entscheidbar, d.h. $K \in \mathcal{L}_0$.

Folgerung

Da das Wortproblem für Typ 1-Sprachen entscheidbar ist, gilt $\mathcal{L}_1 \neq \mathcal{L}_0$.

Beweisidee für den Satz: Berechnung der Funktion χ'_K

- Simulation: gegeben w , simulierte $M_w(w)$
 - falls $M_w(w)$ hält, liefere Wert 1
 - ansonsten terminiert die Simulation nicht, die Funktion ist auf w nicht definiert

□

Die Existenz eines einheitlichen Simulationsalgorithmus ist gesichert:

Satz (UTM-Theorem: Existenz universeller Turing-Maschinen)

Es existiert eine Turing-Maschine $U = M_u$, die die Arbeit jeder beliebigen Turing-Maschine M_w auf jeder beliebigen Eingabe x simulieren kann. Formal:

$$\exists u \forall w, x : M_u(\langle w, x \rangle) \simeq M_w(x).$$

12

Konstruktion einer universellen Turing-Maschine

5.7-8

Beweis

Wir betrachten folgende Maschine $U = M_u$ mit drei Bändern

- *Eingabeband* `band1` und

¹²hierbei steht \simeq für: beide Funktionswerte sind definiert und identisch oder beide sind undefiniert

- *Codierungsband* `band2`,
- *Zustandsband* `band3`.

und dieser Arbeitsweise auf Eingabe $\langle w, x \rangle$:

```

1  # Initialisierung
2  band2 = w # w aus band1 extrahieren und hierher kopieren
3  band1 = x # w (und Trennzeichen etc.) loeschen, Kopf nach links
4  band3 = "Startzustand von M_w" # durch Parsen von w ermitteln
5  # Iteration
6  while (band3 not in "Endzustandsmenge") # durch Parsen von w ermitteln
7      sigma = "Fokuszeichen von Band 1"
8      (q, gamma, move) = delta(band3, sigma)
9      "aktualisiere band1 und Kopf 1 entsprechend (gamma, move)"
10     band3 = q

```

Offenbar lässt sich die Konfigurationsfolge $\text{comp}_{M_w}(x)$ ablesen aus den „band1-Anteilen“ der Konfigurationen, die U jeweils nach Zeile 9 hat.

In diesem Sinne simuliert $\text{comp}_U(\langle w, x \rangle)$ die Berechnung $\text{comp}_{M_w}(x)$. \square

5.7-9 Diskussion

Interpretation

- gewählte Gödelisierung $\text{TM } M \mapsto \text{code}(M) \equiv$ gewählte Programmiersprache (Beispiel: Java-Bytecode)
- $w =$ Quellcode, $M_w =$ Zielcode, $\text{comp}_{M_w}(x) =$ Prozess (mit Eingabe x gestartetes Programm)
- $U =$ Interpreter für Programme in der gewählten Programmiersprache (Beispiel: Java Virtual Machine)

Andere Formalismen

Das UTM-Theorem ist übertragbar auf gleichmächtige Formalismen:

- es gibt ein WHILE-Programm U , das auf Eingabe $\langle P, x \rangle$ den gleichen Wert (inkl. ∞) berechnet, wie das WHILE-Programm P auf x .
- $\exists u \in \text{REK} \forall f \in \text{REK} \exists e \forall x : f(x) \simeq u(e, x)$

„Virtualisierung“

Das Verhalten jeder beliebigen Turing-Maschine T kann durch ein WHILE-Programm „emuliert“ werden, das δ_T (die Übergangstabelle von T) als zusätzliche Eingabe benutzt.

Beispiel: VirtualBox u.a.

5.8 Reduzierbarkeit und Unentscheidbarkeit

Das allgemeine Halteproblem

5.8-1

Folgendes (**allgemeine**) **Halteproblem** für Turing-Maschinen ist unentscheidbar:

$$H = \{w\#x : M_w(x) \text{ terminiert}\}.$$

Beweis: Entscheidung über K auf H zurückführen

- Angenommen, es gäbe einen Entscheider M_H für H .
- Dann berechnet M_H die charakteristische Funktion χ_H :

$$\chi_H(w\#x) = \begin{cases} 1 & \text{falls } M_w(x) \text{ terminiert} \\ 0 & \text{sonst} \end{cases}.$$

- Sei R eine DTM, die $x \mapsto x\#x$ berechnet und M_K die Hintereinanderausführung $R; M_H$.
- M_K berechnet offenbar genau χ_K , d.h. M_K entscheidet K .
- K ist aber nicht entscheidbar, also ist H ebenfalls unentscheidbar. \square

Reduzierbarkeit

5.8-2

Sei $A \subseteq \Sigma^*$ heißt **reduzierbar** auf $B \subseteq \Gamma^*$ (in Zeichen $A \leq_m B$), falls es eine totale und berechenbare Funktion $f : \Sigma^* \rightarrow \Gamma^*$ gibt mit

siehe Paragraph 2.8-5

$$\forall x : x \in A \Leftrightarrow f(x) \in B.$$

Beispiel: $K \leq_m H$ vermittelt $f : w \mapsto w\#w$

Bemerkung

\leq_m steht für „many-one reducible“ und weist darauf hin, dass die Reduktionsabbildung nicht injektiv sein muss: Es ist erlaubt, dass mehrere Eingaben x für Problem A auf die gleiche Eingabe $f(x)$ für Problem B abgebildet werden.

Lemma

Ist B (semi-)entscheidbar und $A \leq_m B$, dann ist auch A (semi-)entscheidbar.

Beweis

- Sei $A \leq_m B$ vermittelt $f : \Sigma^* \rightarrow \Gamma^*$.
- Dann ist $\chi_A = \chi_B \circ f$ (Komposition), denn für alle $x \in \Sigma^*$ gilt:

$$\chi_A(x) = \begin{cases} 1 & , \text{ falls } x \in A \\ 0 & , \text{ falls } x \notin A \end{cases} = \begin{cases} 1 & , \text{ falls } f(x) \in B \\ 0 & , \text{ falls } f(x) \notin B \end{cases} = \chi_B(f(x)) = (\chi_B \circ f)(x).$$

- Die gleiche Aussage gilt für χ'_A , mit ∞ anstelle Wert 0. \square

5.8-3 Das Halteproblem auf leerem Band („ ε -Halteproblem“)

Satz

Die Sprache $H_0 = \{w : M_w(\varepsilon) \text{ hält}\}$ ist unentscheidbar.

Beweisidee: H auf H_0 reduzieren.

Gesucht ist eine totale und berechenbare Funktion f , die jeder Eingabe $w\#x$ für H eine *entscheidungsäquivalente* Eingabe v für H_0 zuordnet, d.h.

$$\forall x\#w : M_w(x) \text{ terminiert} \Leftrightarrow M_v(\varepsilon) \text{ terminiert.}$$

Idee $f : w\#x \mapsto v := \text{code}(N_{w\#x})$, wobei $N_{w\#x}$ eine TM ist, die bei Eingabe ε

1. ihr Eingabeband mit x initialisiert, und den Kopf auf x_1 (Anfang) bewegt
2. sich dann wie $M_w(x)$ verhält

(das Verhalten auf nichtleerer Eingabe ist irrelevant)

Fakt

- $w\#x \in H \Leftrightarrow M_w(x) \text{ terminiert} \Leftrightarrow M_v(\varepsilon) \text{ terminiert} \Leftrightarrow f(w\#x) \in H_0$
- f ist total und berechenbar (Beweis folgt) □

5.8-4 Beweis

Sei v folgender Code, ähnlich dem in der UTM-Konstruktion aus Paragraph 5.7-8

```
1  if band1 == '': # falls Eingabe = leeres Wort ...
2      band1 = x    # <--
3      band2 = w    # <--
4      band3 = "Startzustand von M_w" # durch Parsen von w ermitteln
5  while (band3 not in "Endzustandsmenge") # durch Parsen von w ermitteln
6      sigma = "Fokuszeichen von Band 1"
7      (q,gamma,move) = delta(band3,sigma)
8      "aktualisiere band1 und Kopf 1 entsprechend (gamma,move)"
9      band3 = q
```

Beobachtungen

1. v ist der Code für die Maschine $N_{w\#x}$.
2. v entsteht durch bloßes Einsetzen von x und w in die mit $<--$ gekennzeichneten Stellen in ein entsprechendes „Template“, d.h.
 $f : w\#x \mapsto \text{code}(N_{w\#x})$ ist eine totale und berechenbare Funktion.

Damit ist gezeigt: $H \leq H_0$. □

5.8-5 Ein mächtiges Werkzeug: Der Satz von Rice

Bezeichnung bezeichne f_M die **von TM M berechnete Funktion** $\Sigma^* \rightarrow \Gamma^*$

Wir werden sehen: Es gibt keinen Algorithmus, der für *jedes* w Eigenschaften von f_{M_w} (Semantik) allein aus w (Syntax) ermittelt. Das besagt der

Satz von Rice

Sei $S \subset \text{REK}$ eine *nichttriviale* Teilmenge (d.h. $S \neq \emptyset$ und $S \neq \text{REK}$).

Dann ist folgende Sprache unentscheidbar:

$$\mathcal{C}(S) = \{w : f_{M_w} \in S\}.$$

Beweis durch Reduktion $H_0 \leq_m \mathcal{C}(S)$

□

- Fallunterscheidung nach Lage der überall undefinierten Funktion: (1) entweder $\Omega \in S$ oder (2) $\Omega \notin S$
- im ersten Fall wähle $q \in \text{REK} \setminus S$, im zweiten wähle $q \in S$
- d.h., „ S trennt Ω und q “
- Konstruktion einer TM M'_w , die entweder (1) Ω oder (2) q berechnet
- zeige: $f : w \mapsto \text{code}(M'_w)$ reduziert H_ε auf (1) $\overline{\mathcal{C}(S)}$ bzw. auf (2) $\mathcal{C}(S)$

Die Reduktion (nur für den Fall (1) $\Omega \in S$):

- sei Q TM, die q berechnet
- zu $w \in \Sigma^*$ konstruiere TM M'_w , die auf Eingabe y wie folgt arbeitet:

$$M_w(\varepsilon); Q(y)$$

(d.h., genau dann, wenn $M_w(\varepsilon)$ hält, wird $Q(y)$ gestartet)

- **beachte** $f : w \mapsto \text{code}(M'_w)$ ist total und berechenbar und von $M'_w = M_{f(w)}$ berechnete Funktion g ist

$$g = \begin{cases} \Omega & , \text{ falls } M_w(\varepsilon) \text{ hält nicht} \\ q & , \text{ sonst} \end{cases}$$

- somit gilt $H_\varepsilon \leq \overline{\mathcal{C}(S)}$, denn

$$\begin{aligned} w \in H_\varepsilon &\Leftrightarrow M_w(\varepsilon) \text{ hält, d.h. } M'_w \text{ berechnet } q \\ &\Leftrightarrow \text{von } M_{f(w)} \text{ berechnete Funktion liegt nicht in } S \\ &\Leftrightarrow f(w) \notin \mathcal{C}(S) \end{aligned}$$

- H_ε unentscheidbar $\Rightarrow \overline{H_\varepsilon}$ und somit auch $\mathcal{C}(S)$ unentscheidbar

Diskussion

5.8-6

Konsequenzen

Es gibt keinen Algorithmus, der anhand der Codierung einer Turing-Maschine oder anhand Programmcode entscheidet, ob die berechnete Funktion

- konstant ist
- eine Spezifikation erfüllt
- Schadsoftware enthält
- ...

Beispiel

- $S = \{f \in \text{REK} : f \text{ ist konstant}\}$ ist nichttriviale Menge rekursiver Funktionen
- also ist folgende Sprache unentscheidbar

$$\mathcal{C}(S) = \{w : f_{M_w} \text{ ist eine konstante Funktion}\}$$

5.8-7 „Domino mit veränderten Regeln“

gegeben Folge von Wortpaaren $I = (x_1, y_1), \dots, (x_k, y_k)$ ($x_i, y_i \in \Sigma^+$ für ein beliebiges endliches Alphabet Σ)

gefragt gibt es eine **Lösung**, d.h. eine Folge von Indizes $i_1, \dots, i_n \in \{1, 2, \dots, k\}, n \geq 1$ mit

$$x_{i_1}x_{i_2}\dots x_{i_n} = y_{i_1}y_{i_2}\dots y_{i_n} (= \text{Lösungswort}).$$

Wiederholungen erlaubt

Veranschaulichung

- beliebig viele Exemplare von Dominosteinen der Sorten $\begin{bmatrix} x_i \\ y_i \end{bmatrix}, i = 1, \dots, k$
- Lösung = Aneinanderreihung von Dominosteinen, deren verkettete Oberseiten das gleiche Wort ergeben wie die verketteten Unterseiten

Beispiele

- $(\begin{bmatrix} 1 \\ 101 \end{bmatrix}, \begin{bmatrix} 10 \\ 00 \end{bmatrix}, \begin{bmatrix} 011 \\ 11 \end{bmatrix})$ hat die Lösung $(1, 3, 2, 3)$:

$$\text{Lösungswort: } x_1x_3x_2x_3 = 101110011 = y_1y_3y_2y_3$$

- $(\begin{bmatrix} 001 \\ 0 \end{bmatrix}, \begin{bmatrix} 01 \\ 011 \end{bmatrix}, \begin{bmatrix} 01 \\ 101 \end{bmatrix}, \begin{bmatrix} 10 \\ 001 \end{bmatrix})$ ist lösbar — die kürzeste Lösung besteht aus 66 Indizes!

5.8-8 Das Postsche Korrespondenzproblem

Definition

Das **Postsche Korrespondenzproblem** PCP ist die Menge aller Instanzen I der Aufgabenstellung, für die es eine Lösung $i_1, \dots, i_n \in \{1, 2, \dots, k\}, n \geq 1$ gibt.

PCP ist semi-entscheidbar.

Algorithmus (Brute force)

teste systematisch immer längere Indexfolgen (i_1, i_2, \dots, i_n) auf Gleichheit der verketteten Ober- und Unterseiten

- existiert eine Lösung, so wird eine gefunden

- existiert keine, so terminiert der Algorithmus nicht

Der Satz von Post

5.8-9

Satz (Emil Post, 1946)

PCP ist nicht entscheidbar.

Emil L. Post. A Variant of a recursively unsolvable Problem. Bull. Amer. Math. Soc. 52 (1946), 264-268

Beweisidee

Lemma 1 reduziere allgemeines Halteproblem H auf *modifiziertes PCP*
 $MPCP = \{I \in PCP \text{ hat eine Lösung } (i_1 = 1, i_2, \dots, i_n)\}$

Lemma 2 reduziere $MPCP$ auf PCP

Da H unentscheidbar ist, ist also auch PCP unentscheidbar (\leq_m ist transitiv).

Grobe Beschreibung der Hauptidee für (1)

DTM $(M, x) \mapsto I$ („Dominosteine“ über Alphabet $\Sigma = \Gamma \cup Z \cup \{\#\}$), so dass

- Startkonfiguration $\hat{=} \left[\frac{x_1}{y_1} \right]$ („Startstein“)
- $\delta(z, a) = (z', a', b) \hat{=} \left[\frac{x_i}{y_i} \right]$ geeignet
- und zusätzliche „Hilfssteine“

Ziel: Lösungswort $\hat{=} \text{Berechnung } \text{comp}_M(x)$ (Konfigurationsfolge)

□

Eingeschränkte PCP-Varianten

5.8-10

01PCP = PCP eingeschränkt auf das Alphabet $\Sigma = \{0, 1\}$

PCP_t = PCP eingeschränkt auf $k \leq t$ Wortpaare

boundedPCP • gegeben: PCP-Instanz I , natürliche Zahl L

- gefragt: Existiert Lösung für I der Länge $n \leq L$?

Satz

- $PCP \leq_m 01PCP$,
- PCP_t ist entscheidbar für $t \leq 2$ und unentscheidbar für $t \geq 7$,
- boundedPCP ist *NP-vollständig*.

Mittlerweile verbessert auf $t \leq 5$: T. Neary: Undecidability in Binary Tag Systems and the Post

□

PCP auf Probleme für kontextfreie Grammatiken reduzieren

Correspondence Problem for Five Pairs of Words (2015). 5.8-11

Satz

Für kontextfreie Grammatiken G_1, G_2 , sind folgende Probleme unentscheidbar:

- Ist $\mathcal{L}(G_1) \cap \mathcal{L}(G_2) = \emptyset$? (**Schnittproblem**)
- Ist $|\mathcal{L}(G_1) \cap \mathcal{L}(G_2)| < \infty$?
- Ist $\mathcal{L}(G_1) \cap \mathcal{L}(G_2)$ kontextfrei?
- Ist $\mathcal{L}(G_1) = \mathcal{L}(G_2)$?

Beweisidee (z.B. $PCP \leq_m$ Schnittproblem)

- definiere zu PCP-Eingabe I kontextfreie Grammatiken G_1, G_2 , mit Produktionen (ungefähr) der Form $S_1 \rightarrow$ Oberseite von Stein i , $S_2 \rightarrow$ Unterseite von Stein i (sowie ähnlichen Produktionen, die rechts noch das Startsymbol enthalten)
- überlege, dass *im wesentlichen* gilt: $L(G_1)$ bzw. $L(G_2)$ sind genau die Kombinationen aus Oberseiten bzw. Unterseiten von Dominosteinen
- dann folgt: Es gibt eine Lösung für I genau dann, wenn $L(G_1) \cap L(G_2) \neq \emptyset$
- Trick: Um zusammengehörende Ober-/Unterseiten zu identifizieren, *markiere* sie (und auch die Produktionen) mit der Nummer des Steins

5.8-12 Präzisierung

- $I = ((x_1, y_1), \dots, (x_k, y_k)), x_i, y_i \in \Sigma^*$
- zugeordnete Regelmengende:
 - $G_i = (\{S_i\}, \Sigma \cup \{a_1, \dots, a_k\}, P_i, S_i)$
 - $P_1 = \{S_1 \rightarrow a_1 x_1 | \dots | a_k x_k\} \cup \{S_1 \rightarrow a_1 S_1 x_1 | \dots | a_k S_1 x_k\}$
 - $P_2 = \{S_2 \rightarrow a_1 y_1 | \dots | a_k y_k\} \cup \{S_2 \rightarrow a_1 S_2 y_1 | \dots | a_k S_2 y_k\}$,

wobei die Markierungen $a_1, \dots, a_k \notin \Sigma$ dafür sorgen, dass am erzeugten Wort abzulesen ist, in welcher Reihenfolge die Produktionen angewendet wurden.

- es gilt:

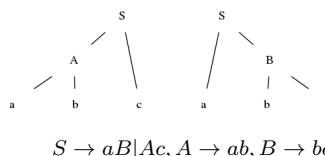
$$\begin{aligned}
 I \text{ besitzt Lösung } i_1, \dots, i_n &\Leftrightarrow a_{i_n} \dots a_{i_1} x_{i_1} \dots x_{i_n} = a_{i_n} \dots a_{i_1} y_{i_1} \dots y_{i_n} \\
 &\Leftrightarrow \mathcal{L}(G_1) \cap \mathcal{L}(G_2) \neq \emptyset
 \end{aligned}$$

Beobachtung

- $L(G_1), L(G_2)$ sind sogar deterministisch kontextfrei (durch DPDA akzeptiert)
- also ist auch das Schnittproblem für DCFL-Sprachen unentscheidbar

Satz

Es ist unentscheidbar, ob eine gegebene Grammatik mehrdeutig ist.

**Beweis**

- reduziere PCP $I = ((x_1, y_1), \dots, (x_k, y_k))$, $x_i, y_i \in \Sigma^*$ auf Mehrdeutigkeitsproblem, d.h. $I \mapsto G$ mit I lösbar gdw. G mehrdeutig
- dazu: $G = (\{S, S_1, S_2\}, \Sigma \cup \{a_1, \dots, a_k\}, P, S)$ mit Produktionen $S \rightarrow S_1|S_2$ sowie
 - $S_1 \rightarrow a_1x_1| \dots | a_kx_k$, $S_1 \rightarrow a_1S_1x_1| \dots | a_kS_1x_k$
 - $S_2 \rightarrow a_1y_1| \dots | a_ky_k$, $S_2 \rightarrow a_1S_2y_1| \dots | a_kS_2y_k$
- i_1, \dots, i_n ist Lösung für I gdw. $w = a_{i_n} \dots a_{i_1} x_{i_1} \dots x_{i_n} = a_{i_n} \dots a_{i_1} y_{i_1} \dots y_{i_n}$ gdw. w lässt sich sowohl über S_1 als auch über S_2 ableiten

Äquivalenzprobleme

- **gegeben** k.f. Grammatiken G_1, G_2 , **gefragt** gilt $L(G_1) = L(G_2)$?
- **gegeben** Kellerautomaten, LBAs, Turing-Maschinen, ... **gefragt** sind die erkannten Sprachen gleich?

Satz

Das Äquivalenzproblem für kontextfreie Sprachen ist unentscheidbar.

Beweisidee reduziere PCP auf das Problem

□

Folgerung

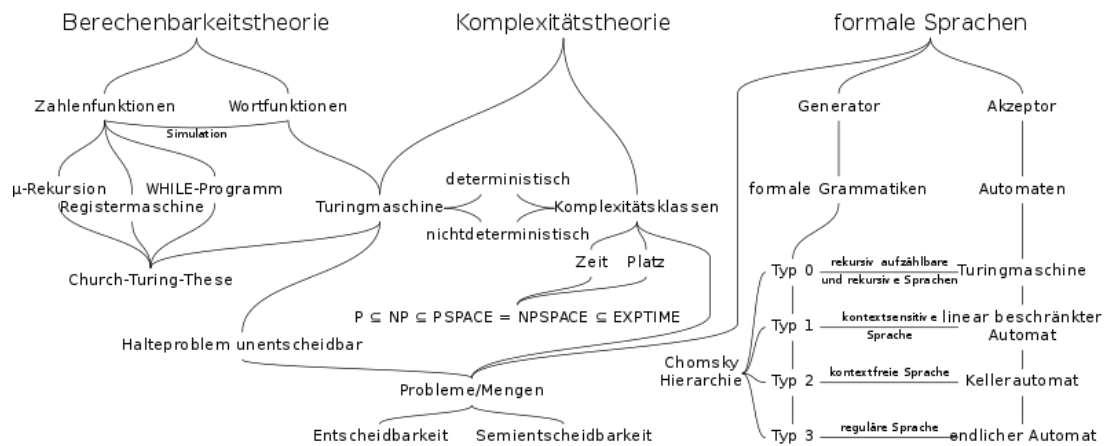
Äquivalenzprobleme für folgende Formalismen sind unentscheidbar:

- Kellerautomaten
- LBAs bzw. kontextsensitive Grammatiken
- Turing-Maschinen
- LOOP-, WHILE-, GOTO-Programme
- ...

denn: k.f. Grammatiken lassen sich *effektiv* in diese Formalismen übersetzen, d.h. es gibt berechnbare Konstruktionen dafür.

5.8-15 Anstelle einer Zusammenfassung ...

... kann ich die Wikipedia-Einstiegsseite zur Theoretischen Informatik (abgerufen am [2020-07-09 Do]) uneingeschränkt empfehlen.



Bildquelle: Wikipedia