Zusammenfassung Formale Sprachen

Felix Lange

13. Dezember 2016

Inhaltsverzeichnis

1 Vokabeln

1.1 Begriffe

Für die nachfolgenden Konzepte sind folgende Vokabeln Essentiell:

- **Alphabet** – Endliche Menge an Zeichen
- **Wort** – Folge von Zeichen aus einem Alphabet
- **Länge** – Anzahl der Zeichen in einem Wort
- **leeres Wort E** – das leere Wort, also keine Eingabe
- **formale Sprache** - Formale Sprache L ist Teilmenge eines Alphabetes, oder auch über das gesamte Alphabet gespannt
- **E-freier Konkatenationsabschluss** - $L^+ = \cup_{k=1} L^k$, Menge aller Wörter, die man aus einem oder mehreren Wörtern aus L schreiben kann.
- **Konkatenationsabschluss** - $L^* = \cup_{k=0} L^k = L^0 \cup L^+ = \{E\} \cup L^+$

1.2 Operationen

Des weiteren sind folgende Operationen über Wörter, Alphabete etc. essentiell:

- **Konkatenation** - $w_1 * w_2 = w_1 w_2$, also einfaches aneinanderhängen der wörter **Vorsicht, nicht kommutativ!**
- **Potenz** - die Potenz eines leeren Wortes ist E, ansonsten gilt $w^{k+1} = w^k w$
- **Konkatenation einer Sprache** - $l_1 * l_2 = l_1 l_2 = \{w_1 w_2 | w_1 \in l_1 w_2 \in l_2\}$

2 Anwendungen von Sprachen

2.1 Deterministischer Endlicher Automat

endlicher Automat - festgelegt durch:

- endliche Zusandsmenge Z
- einen Anfangszustand $z_0 \in Z$
- ein Eingabealphabet
- eine Überführungsfunktion $f : Z \times X \to Z$
- eine Überführungsfunktion für folgen von Eingabezeichen $f^*(z, E) = z$, $f^*(z, wx) = f(f * (z, w), x)$
- eine Überführungsfunktion, um alle Zustande auf dem weg durch eine Eingabe zu erhalten: $f^{**}(z, E) = z$, $f^{**}(z, wx) = f^{**}(z, w) f(f^*(z, w), x)$
- Eine optionale Ausgabefunktion:

  Mealy-Automat, zu jeder Eingabe auch eine $Ausgabe : Z \times X \to Y$

  Moor-Automat, zu jedem Zustand eine $Ausgabe : Z \to Y$

Ein **Akzeptor** ist ein Automat mit einer Menge F an akzeptierenden Zuständen, die eine Teilmenge von Z ist und akzeptierte Eingaben kennzeichnet. Die von einem Akzeptor M **akzeptierte Sprache** ist definiert als $L(M) = \{winX^* | f^*(z_0, w) \in F\}$

2.2 Reguläre Ausdrücke

2.2.1 Definition

Sei A ein Alphabet das die Zeichen aus Z, $Z = \{|, (,), *, \emptyset\}$, nicht enthält. Dann ist ein **Regulärer Ausdruck** über A eine Zeichenfolge über dem Alphabet $A \cup Z$, die gewissen Vorschriften genügt. Die menge der Regulären Ausdrücke ist wie folgt festgelegt:

- $\emptyset$ ist ein regulärer Ausdruck
- für jedes $a \in A$ ist a ein regulärer Ausdruck
- sind R_1 und R_2 reguläre Ausdrücke, so auch $(R_1|R_2)$ und (R_1R_2).
- ist R ein regulärer Ausdruck, so auch $(R*)$
- Nichts anderes sind reguläre Ausdrücke.

Dabei können die Klammern weggelassen werden, es gilt dabei folgende Mächtigkeit: $Stern > Punkt > Strich$!
Die von einem regulären Ausdruck R beschriebene Sprache $< R >$ is wie folgt definiert:

- $< \emptyset >= \emptyset = \{\}$
- für $a \in A$ ist $< a >= \{a\}$
- sind R_1, R_2 reguläre Ausdrücke, so gilt:
  $< R_1|R_2 >=< R_1 > \cup < R_2 >$
  $< R_1R_2 >=< R_1 > . < R_2 >$
- ist R ein regulärer Ausdruck, so ist $< R* >=< R >^*$

Hat man 2 Reguläre Ausdrücke und möchte nachweisen dass sie das selbe bewirken, so ist das Prinzipiell möglich, dabei liegt aber die Schwierigkeit im Raum von PSPACE, d.H. die Lösung dauert extrem lange!

2.2.2 erzeugung

Ein Regulärer Ausdruck kann wie folgt aus einem Akzeptor erzeugt werden: Wir definieren zunächst die formalen Sprachen

für k = 0: $L_{ij}^k = \{w \in X^* | f^{**}(z_i, w) \in z_i Z_k^* z_j \vee (i = j \wedge w = E)\}$.

für k > 0: $L_{i,j}^k = L_{i,j}^{k-1} \cup L_{i,k-1}^{k-1}(L_{k-1,k-1}^{k-1})^* L_{k-1,j}^{k-1}$

Dabei ist i der Startzustand, j der Zielzustand und k der maximale Knotenindex auf dem Weg zu j.
Vorgegangen wird nun wie folgt:

1. Die Sprache kann mit Hilfe von Vereinigungen diverser L_{ij}^k ausgedrückt werden.
2. Dabei kann von k=0 ausgehend das k solange erhöht werden, bis k = Anzahl der Knoten ist.
3. Die nun erhaltene Sprache lässt sich in einen regulären Ausdruck umwandeln, dieser Schritt ist bereits bei Teillösungen (Sprachen) Hilfreich.

2.2.3 Operationen

Ausschluss einer Sprache aus einem Alphabet: Ausschließen der Aktzeptierenden Zustände der Sprache aus der Menge der Akzeptierenden Zustände des neuen Akzeptors.

Der Durchschnitt wird als paralleles verarbeiten der Akzeptoren betrachtet und ist als neuer Akzeptor implementierbar. dabei gilt:

- im neuen Akzeptor gibt es z1 x z2 viele Zustände, also zu jedem Zustand des eines Akzeptors jewails nochmal alle Zustände des anderen Akzeptors.
- die Übergangsfunktion ist definiert als $f((z_1, z_2), a) = (f_1(z_1, a), f_2(z_2, a))$, also als abarbeiten beider Akzeptoren parallel.

- die Menge der Akzeptierenden Zustände F ist: $F = F_1 \times F_2$, also Akzeptiert der neue Akzeptor nur, Wenn Akzeptor 1 **und** 2 akzeptieren!
- $z_0 = (z_{01}, z_{02})$.

2.3 Grammatiken

eine Gramatik $G = (N, T, S, P)$ ist definiert durch

- Einem Alphabet der nicht-terminale, **N**
- Ein Alphabet **T** aus Terminalsymbolen, dijunkt zu N
- Einem Startsymbol **S** aus N
- Eine Menge sog. Produktionen, $\mathbf{P} \subset V^*NV^* \times V$, $V = N \cup T$. Die Produktionen werden häufig in der form $v \to w$ notiert. links ist dann ein Nichtterminal, rechts ein Element aus V oder das leere Wort.

Zur überprüfung ob ein Wort in der Gramatik beschrieben ist wird durch einsetzetn der Produktionsregeln und stückweisem ersetzen geprüft, ob das Wort mithilfe der Produktionsregeln erzeugt werden kann. Ein übergang von einem Wort zu einem Anderen durch einsetzen der Produktionsregeln wird gekennzeichnet als $\to$.
Die von einer Grammatik $G = (N, T, S, P)$ erzeugte Sprache ist:
$L(G) = \{w \in T^* | s \to *w\}$

2.4 Typ-3-Grammatiken

Eine Typ-3-Grammatik genügt folgenden Einschränkungen: Jede Produktion ist entweder von der Form $X \to w$ oder $X \to wY$ mit $w \in T^*$ und $X, Y \in N$.
Auf der Rechten Seite darf also höchstends nur ein Nichtterminalsymbol vorkommen, und wenn dann nur als letztes Symbol. Das is tdefiniert als **rechtslineare Grammatik**.

2.5 Nichtdeterministische endliche Automaten

2.5.1 Definitionen

Eine **Potenzmenge** einer Menge Z ist die Menge aller möglichen Teilmengen von Z, sie wird als 2^z bezeichnet.

ein **nichtdeterministischer endlicher Akzeptor** $N = (Z, z_0, X, G, F)$ (kurz NEA) ist festgelegt durch:

- eine endliche Zustandsmenge Z
- einen Anfangszustand $z_0 \in Z$
- ein Eingabealphabet X
- eine menge $F \subseteq Z$ akzeptierender Zustände
- eine Funktion $g : Z \times X^* \to 2^Z$ mit der Eigenschaft, dass nur für endliche viele Paare gilt: $g(z, w) \neq \emptyset$

Bei einem NEA kann ein Zustand für die selbe Eingabe mehrere verschiedeneNachfolgezustände haben, bei denen zuvor nicht klar ist in welchen der Automat springen wird. Allerdings bedeutet ein Fehlender abgehende Pfeil von einem knoten hier auch, dass der NEA keine möglichkeit hat die Eingabe zu verarbeiten und sie daher nicht akzeptieren - selbst wenn alle Zustände des NEA akzeptierend sind.
Für die Zustände $z_1, z_2 \in Z$ eines NEA und ein $w \in X$ schreibt man $z_1 \to^w *z_2$, falls

- $z_1 = z_2$ und w = E ist oder
- es ein $z\prime \in Z$ und Wörter $w\prime, w\prime\prime \in X^*$ gibt mit $w = w\prime w\prime\prime$ und $z_1 \to^{w\prime} *z\prime \to^{w\prime\prime} z_2$

Damit kann die Funktion $g^* : Z \times X^* \to 2^Z$ durch die Forderung $g : Z \times X^* \to 2^Z$ festgelegt werden.

die von einem NEA N **erkannte Sprache** ist die Menge
$L(N) = \{w \in X^* | g^*(z_0, w) \cap F \neq \emptyset\}$
Aller wörter, bei deren Eingabe der Automat vom Startzustand in einen akzeptierenden zustand übergehen kann.

2.5.2 überführung NEA zu DEA

Die Überführung beruht auf folgenden, groben Schritten:

- Beseitigen von Übergängen zwischen Zuständen mit einer Wortlänger über 1
- Beseitigen der Zustandsübergänge die das leere Wort verwenden. I.d.R ändert das auch die Menge akzeptierender Zustände
- Beseitigen von Situationen, die echt nichtdeterministisch sind, also wo mindestends 2 Nachfolger für die selbe Eingabe existieren. Dabei werden übergänge, die in das nnichtsßeigen, auch gelöst.

Für den letzten Schritt wird dabei etwas komplexer gearbeitet:

- Dazu wird ein Neuer Automat erzeugt, der sich nach einem nichtdeterministischen Zustand alle Zustände merkt, die der Automat ab dort annehmen kann. im weiteren wird jede der Möglichkeiten weitergeführt. Dafür wird die Zustandsmenge zunächst auf 2^{Z_n} Zustände erhöht, d.H. alle möglichen Teilmengen aus Z werden mit eingeschlossen.
- Der Neue Automat sollte akzeptieren, wenn eine der Möglichkeiten in einen Akzeptierenden Zustand führt
- die Nachfolgerzustände werden immer aus dem alten Automaten generiert, auch bei zuständen die Teilmengen darstellen.

2.6 Zustandsminimierung endlicher Automaten

2.6.1 Definitionen

Bevor wir die Zustände trennen können sind noch Definitionen nötig:

- eine **Äquivalenzrelation** (Zeichen $\equiv$) ist eine Relation $R \subseteq M \times M$ falls gilt:

  R ist reflexiv, d.H für alle $x \in M$ gilt: $(x, x) \in R$

  R ist Symmetrisch, d.h. für alle $x, y \in M$ gilt wenn $(x, y) \in R$ auch $(y, x) \in R$

  R ist transitiv, d.H. für alle $x, y, z \in M$ gilt wenn $(x, y), (y, z) \in R$ auch $(x, z) \in R$
- **getrennte wörter** sind Wörter, die der Automat unterscheiden können muss. Das führt zu der Überlegung, was sich der Automat alles merken können muss und ob Manche unterscheidungen in Form von Zuständen evtl. Überflüssig sind.
- eine **Äquivalenzklasse** ist eine Unterteilung einer Menge M anhand einer Äquivalenzrelation in Teilmengen von M. Jedes Zeichen aus M liegt in genau nur einer Äquivalenzklasse, geschrieben [x]. Die Äquivalenzklassen umfassen also alle Elemente aus M, die nach einer bestimmten Relation gleich sind. Die Menge aller Äquivalenzklassen schreibt mann als $M/_{\equiv}$

2.6.2 Vorgehen

Für die Zustandsminimierung endlicher Automaten, die an sich beliebig groß sein können (was aber seltendst wirklich erwünscht ist) wird wie folgt vorgegangen:

- Entfernen aller nicht erreichbarer Zustände
- Anwenden eines Algorithmusses, der die Zustände in möglichst wenige Äquivalenzklassen unterteilt. Die Klassen untereinander müssen sich dann unterscheiden, die Elemente einer Klasse sind aber immer Äquivalent. Vorgegangen wird so:

  der Algorithmus beginnt mit einem kleinen $\equiv_j$, bsp j = 0, und erhöht j sukzessiv. Dabei bezeichnet j die Wortlänge der Eingabe. 2 Zustände sind gleich, wenn sie nach Eingabe von Wörtern der maximalen Länge j nicht anhand der Ausgabe (akzeptiert oder nicht) unterscheidbar sind. die erhöhung der j endet erst, wenn sich an den Äquivalenzklassen nichts mehr ändert!

3 Regular Expressions

Die Regular Expressions R mmatchenëin Wot w, falls das wort durch R beschrieben werden kann, d.h. gewisse Vorschriften befolgt.

3.1 GNU egrep

Egrep existiert in mehreren implementierungsvarianten, im folgenden wird GNU egrep verwendet. Die generelle Funktionsweise is **egrep REGEX DATEINAME**. Dabei wird der Teil des Regex, der nur zeichen (also inhalt) des gesuchten Substrings enthält in Single-quotationmarks gesetzt (´´). Weitere Zeichen Sind:

$'TEXT'$	Text kennzeichnen
|	Alternativen
$*$	Beliebig oft wiederhohlt, auch 0 mal
()	Gruppiert
$+$	mindestends ein mal
?	entweder kein- oder einmal
E	statt der leeren Menge
[a...z] oder [a - z]	ersatz für $a\|b\|...\|z$, ohne Komma, genannt **Zeichenklasse**
[^ a...z]	negierte Zeichenklasse: Alle Zeichen, außer die in der Zeichenklasse.
R{n}	R wird n mal wiederhohlt
R{min,max}	R wird minimal min mal wiederhohlt, maximal max mal. Ist max leer wird es nicht gewertet
.	steht für ein beliebiges Zeichen

Die genannten Steuersymbole müssen escaped werden, falls sie als normale Zeichen gesucht werden. Dafür gelten die folgenden Regeln:

- ein - unmittelbar am Anfang einer (negierten) zeichenklase steht für sich als Zeichen
- ein ^ das nicht unmitelbar nach [kommt steht für sich als Zeichen
- ein] direkt nach einem [oder [^ steht für sich als Reguläres Zeichen
- eine [nach einem [oder [^ steht für sich als Zeichen

Um Zeichenfolgen nur am Anfang oder am Ende zu untersuchen werden sogenannte Anker verwendet: ^ ist für den Anfang einer zeichenfolge und $ für das Ende einer Zeichenkette. Die Ankwer werden entsprechend links bzw rechts vom Suchstring positioniert.

3.1.1 Gruppierung und mehrere Treffer

Sind in einem Wort w mehrere Substrings die dem Regex R entsprechen, wird immer das Erste auftreten bevorzugt. die länge das gematchten Textes hängt dabei von der Spezifikation ab:

- nach POSIS Spezifikation: bei gleichem Startpunkt im String wird der längere match immer bevorzugt.
- Bei anderne (Perl) haben kompliziertere Verfahren, die dafür aber schnellere Verfahren bieten.

4 Typ-2

4.1 Kellerautomaten

Kellerautomaten sind eine Verallgemeinerung der endlichen Akzeptoren, die eine höhere mächtigkeit bieten sollen um Formale Sprachen, de nicht als DEA darstellbar sind, zu verarbeiten. Dabei haben kellerautomaten eine unterschiedliche mächtigkeit je nachdem, ob sie deterministisch sind oder nicht, anders als DEA / NEA. Als Änderung gegenüber DEA / NEA besitzen die Kellerautomaten **Speicher**. Dieser ist theoretisch unendlich, wird aber zunächst eingeschränkt. Speicherkann dabei nach Größe oder nach zugriffsmöglichkeit beschränkt werden, letzteres erfolgt in den Folgenden Modellen. Die Zugriffsbeschränkungen können sich wie folgt verhalten: entweder es steht nur ein Zähler bereit, oder es kann nur auf das zuletzt abgespeicherte zugegriffen werden. Weitere Beschränkungsformen sind aber auch möglich.
Ein Wort gilt in einem KA dann als akzeptiert, wenn nach einiger zeit alle Eingaben gelesen sind und die Steuereinheit in einem akzeptierenden Zustand ist.

4.1.1 Definition NKA

Ein **nichtdeterministischer kellerautomat NKA** besteht aus:

- endliche Steuereinheit - stets in einem von endlich vielen Zuständen
- Eingabeband - enthält schrit für schrit eingelesene Eingabesymbole
- keller(Speicher) - erlaubt nur zugriff auf oberstes Element
- einer endlichen Zustandsmenge (formal)
- einem Anfangszustand (formal)
- ein Kelleralphabet (formal)
- ein Anfangskellersymbol (formal)
- ein Eingabealphabet (formal)
- eine Überführungsfunktion (formal)
- eine menge Akzeptierender zustände (formal)

Ein NKA kann sich beim Einlesen entscheiden, ob er einlesen möchte oder nicht, ob er speichert oder nicht und welchen zustand er nun wählt. Daher **nicht deterministisch**

4.1.2 Definition DKA

Ein **deterministischer kellerautomat DKA** ist im prinzip wie ein NKA definiert, muss aber Bedingungen erfüllen:

- Es muss eindeutig festgelegt werden, ob in einem bestimmten zustand bei einer bestimmten Eingabe gelesen wird oder nicht.
- wird in einem zustand nicht gelesen, gibt es nur eine Aktion für alle Eingaben
- wird gelesen, so gibt es für jede Eingabe nur eine möglichkeit fortzufahren

4.1.3 Sprache eines Kellerautomaten

Die von einem Kellerautomaten K erkannte Sprache ist die menge der Eingabewörter, die bei Eingabe in K mit frisch initialisierten Speicher eingegeben wird und dabei als Endzustand einen akzeptierenden Zustand aufweist.

4.1.4 Spiegelung

Für ein Wort $w \in A$ bezeichne w^R das Spiegelbild von w. Dabei gilt des weiteren: $E^R = E$ und für $x \in A, w \in A^*$ sei $(xw)^R = w^R x$

4.2 kontextfreie Grammatiken

4.2.1 Definition

Eine kontextfreie Sprache besitzt nur Produktionsregeln, an deren linken Seite nur ein einziges Nichtterminalsymbol steht. Ein alternativer Name ist dabei Typ-2-Grammatiken (T2G), eine Formale Sprache die von einer kontextfreien Grammatik aufgespannt wird wird als kontextfreie Sprache bezeichnet.

4.2.2 begriffe

Linksableitungsschritt - Ableitungsschritt, der das am weitesten linke nicht-terminalsymbol ersetzt. Geschrieben wird: $\rightarrow^l$

Linksableitungsfolge - eine Folge von Ableitungen die nur aus Linksableitungen besteht. man Schreibt: $\rightarrow^{l*}$

Rechtsableitungsschritt - Analog zum Linksableitungsschritt!

Rechtsableitungsfolge - Analog zur Linksableitungsfolge!

Ableitungsbaum - ein Ableitungsbaum zu einer Ableitung $X \rightarrow^* w$ ist ein Graph, dessen knoten mit nicht-terminalsymbolen, Terminalen oder E beschriftet sind. Der Aufbau ergibt sich wie folgt:

- Wurzel des Baumes mit X beschriftet
- Blätter von links nach rechts aneinandergereiht ergeben w
- wird ein nicht-Terminal ersetzt, so wird ihm ein nachfolgerknoten mit dem eingesetzten Terminal angehängt.
- ein Ableitungsbaum von mehreren Ableitungen ist möglicherweise gleich.
- zu einem Ableitungsbaum gibt es aber nur genau eine Linksableitung und genau nur eine Rechtsableitung.
- Daher: Jede Ableitung hat eine äquivalente links- bzw. Rechtsableitung!

Mehrdeutigkeit - gibt es für ein Wort mindestends 2 verschiedene Ableitungsbäume (also auch 2 verschiedene linksableitungen) so ist die Sprache mehrdeutig, sonst ist sie eindeutig.

(inhärente) mehrdeutigkeit einer Sprache - eine kontextfreie Sprache ist mehrdeutig, falls jede sie erzeugende T2G mehrdeutig ist.

4.3 Zusammenhang von Kellerautomaten und T2G

4.3.1 Kellerautomaten beim Arbeiten mit T2-Gramatiken

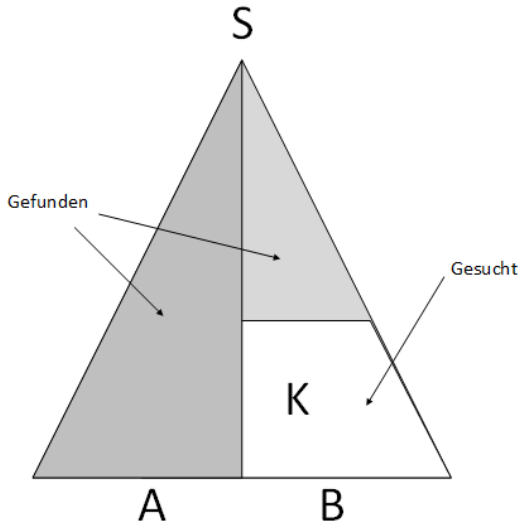
Eine formale Sprache kann genau dann von einem Kellerautomaten erkannt werden, wenn sie von einer T2G erzeugt werden kann. Zur erkennung geht ein Automat für eine T2G Grammatik vor wie folgt:

1. ohne lesen zunächst X0 kellern
2. Wenn oberstes Symbol nicht-Terminal: Ersetzten
3. Wenn oberstes Symbol Terminal: als eingabe Lesen, aus keller entfernen und weiter lesen
4. wird am ende in das Initialisierungssymbol im Speicher las einziger inhalt gefunden, so wird akzeptiert.

Die Ersetz- oder Leseschritte mit den neuen Zuständen sind dabei immer eindeutig definiert, der Automat ist aber nicht Deterministisch, da unter umständen mehrere Ersetzungsschritte möglich sind. Daher könnte der Automat bei manchen ERsetzungsfolgen in einen falschen Endzustand wechseln!

4.3.2 Top-Down-Syntaxanalyse

Konstruiert der Automat den Ableitungsbaum von Oben nach Unten, so wird von einer Top-Down-Analyse gesprochen. Sei A der bereits gelesene Teil der Eingabe, W der ausstehende Teil der Eingabe und K der kellerinhalt, so lässt sich die Situation vor einem Ersetzungsschritt wie folgt veranschaulichen:



Der Graue Teil wurde dabei schon gefunden, der weiße Teil noch nicht, es gilt also $S \rightarrow^{l*} Ak$. Bei Top-Down erzeugt der Kellerautomat also stets eine Linksableitung.

4.3.3 Umsetzung in der Realität

Da die nicht-deterministischen Kellerautomaten in der Realität nicht in deterministischen Computern umsetzbar sind, müssen andere Lösungen gefunden werden:

- Simulation durch Suche nach akzeptierender Berechnung, dabei muss aber aufgepasst werden dass sich der Rechner nicht in einer unendlichen Rekursion befindet. Auch ist der Zeitaufwand gegebenenfalls zu hoch.
- Man schränkt den Kellerautomaten so ein, dass im Extremfall nur noch die deterministischen möglichkeiten zugelassen werden. Alternativ kann auch die Definition eines Kellerautomaten aufgeweicht werden, sodass der Automat mehr als nur das oberste Zeichen des Speichers sehen kann.
- Als letzte möglichkeit kann das Konzept der Kellerautomaten auch verworfen werden und die Lösung über einen anderen Algorithmus implementiert werden, siehe dazu kapitel 6.

4.3.4 modifizierter Kellerautomat für bottom-up Analysen

- Ein praktischer Ansatz ist anzunehmen, dass der Kellerautomat auch mehr als ein zeichen des Speichers lesen könnte. umsetzbar ist das z.B. durch eine erhöhung der zustandszahl und das Speichern der Daten in der Steuereinheit. Im folgenden wird immer davon ausgegangen, dass der Automat dies beherrscht.
- Davon ausgehen, dass das oberste Element im Speicher das letzte Teil des Wortes ist. im folgenden wird aber immer nur ein Symbol gekellert, ein vertauschen der Richtung ist also ausgeschlossen.

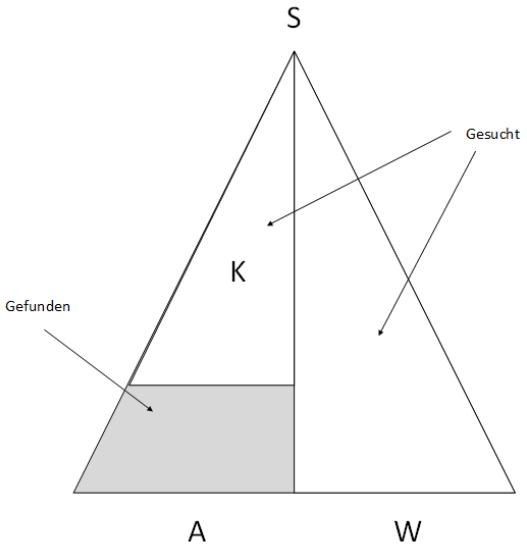
4.3.5 Bottom-Up Analyse

Bei Bottom-Up wird der Automat vorgehen wie folgt:

1. Noch nicht gelesene Eingabe ist das zu beweisende Wort
2. Nacheinander wird der inhalt gelesen und gekellert (Leseschritt)
3. Sobald ein Ausdruck durch ein Nicht-Terminal ausgedrückt werden kann wird das nicht-Terminal eingesetzt. (Reduktionsschritt)

4. falls möglich werden auch Nicht-Terminale mit Terminalen zusammen weiter zusammengefasst(Reduktionsschritt)
5. falls nach dem Ende der eingabe der verbleibende Kellerinhalt auf das Startsymbol reduziert werden kann wechselt der Automat in den akzeptierenden Zustand.

Bei der Bottom-up analyse gilt also: $k \rightarrow^{r*} A$, grafisch veranschaulicht:



Bei der Bottom-Up Analyse hat der Kellerautomat dabei einige Freiheitsgrade:

- Es kann sowohl ein lese- asl auch ein Reduktionsschritt gleichzeitig möglich sein
- bei Reduktionsschirtten können unterschiedlich lange Teile reduziert werden
- Bei einem Reduktionsschritt kann aus mehreren Alternativen mit gleicher rechter Seite ausgwählt werden

Für einen Deterministischen Automaten muss die Sprache also geeignet Konstruiert werden.

5 Typ 1 und Typ 2

5.1 Kontextsensitive Grammatiken

5.1.1 Definition

Eine Typ-1-Grammatik (T1G) oder auch kontextsensitive Grammatik ist eine Grammatik G = (N, T, X0, P), deren Produktionen alle von einer der folgenden Formen sind:

- uXv $\rightarrow$ uwv mit $u, v \in V^*, w \in V^+ und X \in N$ oder
- X0 $\rightarrow E$. Falls diese Produktion existiert, kommt aber X0 in keiner Produktion auf der rechten Seite vor.

Eine formale Sprache ist vom Typ 1 oder kontextsensitiv, wenn es eine T1G gibt, die sie erzeugt.

5.1.2 Eigenschaften

Im Beispiel kann X durch ein Wort w ersetzt werden, aber im Unterschied zu kontextfreien Grammatiken ist das nicht immer moglich, sondern nur, wenn das X in einem gewissen Kontext vorkommt: links von X muss u stehen und rechts davon v, sonst ist die Produktion nicht anwendbar.
Jede kontextfreie Sprache ist kontextsensitiv!
Mit kontextsensitiven Grammatiken kann man also echt mehr formale Sprachen erzeugen als mit kontextfreien.

5.2 Turingmaschinen

Die Turingmaschine stellt eine Erweiterung der Kellerautomaten dar, bei der der Keller durch einen Speicher mit Wahlfreiem Zugriff ersetzt wird.

5.2.1 Definition

Eine Turingmaschine (TM) besteht aus einer endlichen Steuereinheit und einem unendlichen Arbeitsband, das zeichen des Bandalphabetes speichert. Uber einen Schreib-Lese-Kopf hat die TM Zugriff auf jeweils ein Feld des Bandes, der Kopf kann in jedem Schritt um ein Feld weiter geruckt werden. Formal ist eine TM definiert durch:

- eine endliche Zustandsmenge Z,
- einen Anfangszustand z0 aus Z,
- ein Bandalphabet Y,
- ein Blanksymbol aus Y,
- ein Eingabealphabet X $\subseteq$ Y,
- eine nichtleere Menge F + $\subseteq$ Z akzeptierender Endzustande und eine nichtleere Menge F $\subseteq$ Z ohne F+ ablehnender Endzustande und
- eine Uberfuhrungsfunktion f : Z Y Z Y 1, 0, 1 .

5.2.2 Eigenschaften

Wenn eine TM einen (akzeptierenden oder ablehnenden) Endzustand erreicht hat, dann verlaßt sie ihn nicht mehr, andert nicht mehr die Bandbeschriftung und bewegt den Kopf nicht mehr. Die TM halt an.

Bei der Uberfuhrungsfunktion bedeutet f (z, y) = (zf , yf , d), dass die TM, wann immer sie sich in Zustand z befindet und auf dem Band Symbol y liest, in Zustand zf ubergeht, auf das Band Symbol yf schreibt und den Kopf um d Felder nach rechts bewegt.

Formal ist das Band zweiseitig unendlich. Aus oben Gesagtem ergibt sich aber, dass zu jedem Zeitpunkt nur ein endlicher Abschnitt interessant“ ist, in dem nicht alle Felder mit leer beschriftet sind. Und wenn eine Turingmaschine nach endlich vielen Schritten halt, dann hat sie nur endlich viele Felder besucht. Der tatsachlich benutzte Speicherbereich ist also endlich.

5.2.3 Verbindung zu formalen Sprachen

Eine formale Sprache L heißt rekursiv, wenn es eine TM gibt, die fur jedes w ∈ L als Eingabe irgendwann in einen akzeptierden Endzustand ubergeht und fur jedes w ∉L als Eingabe irgendwann in einen ablehnenden Endzustand ubergeht.

Eine formale Sprache L heißt rekursiv aufzahlbar, wenn es eine TM gibt, die fur jedes w ∈ L als Eingabe, und auch nur fur diese Worter, irgendwann in einen akzeptierden Endzustand uber- geht. Fur Eingaben w ∉ L wird nur gefordert, dass sie nicht akzeptiert werden; die TM darf aber irgendwann in einen ablehnenden Endzustand ubergehen oder unendlich arbeiten ohne je zu halten.

Wenn eine Sprache L rekursiv ist, dann ist auch ihr Komplement Lk rekursiv.

Eine formale Sprache kann genau dann von einer Typ-0-Grammatik erzeugt werden, wenn sie rekursiv aufzahlbar ist.

5.2.4 intuitiv berechenbar

Ein Algorithmus kann dadurch definiert werden, dass

- eine endliche Beschreibung vorliegen hat, die in
- elementare Schritte zerfallt, die offensichtlich intuitiv berechenbar sind.
- Die Abarbeitung des Algorithmus soll schrittweise erfolgen und
- nach jedem Schritt soll eindeutig der als nachstes durchzufuhrende Schritt festliegen.
- Der Algorithmus soll fur beliebig große Eingaben (also unendlich viele) die richigen Ergebnisse produzieren.
- Der Algorithmus soll fur jede Eingabe nach endlich vielen Schritten halten.

Wenn eine Funktion durch einen Algorithmus dieser Art berechnet werden kann, konnte man die Funktion intuitiv berechenbar nennen. Dies ist aber offensichtlich keine mathematisch harte Definition des Berechenbarkeitsbegriffes.

These von Church-Turing: Alles, was intuitiv berechenbar ist, ist auch von einer Turingmaschine berechenbar.

5.2.5 Beschränkungen der Touringmaschine

Eine Turingmaschine ist t (n)-zeitbeschrankt, wenn fur alle $s \in$ gilt, dass sie fur Eingabeworter w der Lange n hochstens t (n) Schritte macht bis sie anhalt.

Eine Turingmaschine ist s(n)-raumbeschrankt oder s(n)-platzbeschrankt, wenn fur alle $n \in$ gilt, dass sie fur Eingabeworter w der Lange n hochstens s(n) Felder auf dem Arbeitsband besucht bis sie anhalt.

Ein linear beschrankter Automat ist eine Turingmaschine, die n + 2-platzbeschrankt ist. M. a. W. konnen also nur die n Felder, auf denen die Eingabesymbole stehen und (um die Enden der Worter zu erkennen) jeweils das erste Feld links und rechts daneben besucht werden.

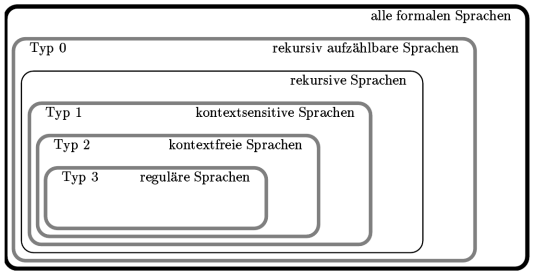
Wächst der Speicherbedarf oder zeitbedarf eines Algoritmuses schneller als der eines anderen Algorithmuses, so sagt man auch das dieser Algorithmus echt mehr formale Sprachen erkennen kann. Das bedeutet zum Beispiel, dass Turingmaschinen, die fur Worter der Lange n Speicherplatzbedarf n^2 haben, echt mehr rekursive formale Sprachen erkennen konnen als Turingmaschinen, die Speicherplatzbedarf n + 2 haben.

5.2.6 nicht rekursiv aufzählbare Sprachen

Es gibt formale Sprachen, die nicht rekursiv aufzahlbar sind. Beispiele Folgen.

5.2.7 Chomsky-Hierarchie

Alle Inklusionen in der Abbildung sind echt: Es gibt immer eine (sogar unendlich viele) formale Sprachen, die in einer Klasse enthalten sind, aber nicht in der nachst kleineren.



6 Syntaxanalyse

6.1 Lexikalische Analyse

Ein Programm(teil), das die lexikalische Analyse durchfuhrt, heißt auch Lexer oder Scanner. Im Deutschen benutzen manche den Begriff Symbolentschlussler.

Aufgabe der lexikalischen Analyse ist, den Eingabetext an den passenden Stellen in eine Folge sogenannter Lexeme aufzuteilen. Jedes Lexem passt zu einem Muster (reguläre Ausdrücke).

Ein Lexer stellt neben Prozeduren zur Initialisierung und Beendigung als Schnittstelle zur syntaktischen Analyse eine Prozedur bereit, die im folgenden stets nextToken heißen soll. Ein Aufruf fuhrt dazu, dass der Lexer im Eingabestrom nach dem nachsten Lexem sucht, fur das er ein sogenanntes Token zuruckzuliefern hat. Ein Token besteht aus einem Typ und unter Umstanden noch weiteren Informationen, den sogenannten Attributen des Tokens.

Jeden Kommentar kann man als Lexem auffassen, fur das kein Token erzeugt wird. Ebenso sollte der Lexer alle Leerzeichen uberlesen, jedenfalls insoweit sie fur das Programm bedeutungslos sind.

Beispiel: Eingabe erna = hugo + 1; nextToken Ergebnis: (id,erna) (assignOp,) (id,hugo) (plusOp,) (num,1) (semi,)

6.2 Lexer-Generatoren

Es gibt Programme zur automatischen Erzeugung von Lexern, sogenannte Lexer-Generatoren. Als Beispiele seien lex, flex und JLex genannt, aber auch andere Ubersetzergenerator-Suiten wie cocktail und Eli enthalten solche Werkzeuge. Lexergeneratoren erhalten als Eingabe eine Datei, in der insbesondere mit Hilfe regularer Ausdrucke beschrieben ist, Lexeme welcher Struktur zu suchen sind, und welche Token daraus zu erzeugen sind. Als Ausgabe erzeugt ein Lexergenerator die Quelle eines Programmes, das als Lexer auf die gewunschte Art arbeitet.

Der besseren Lesbarkeit wegen erlaubt man, die Beschreibung eines regularen Ausdrucks zu strukturieren, indem man Teilausdrucke mit Namen bezeichnen kann. Wir wollen das als regulare Definitionen bezeichnen, Vorgehen und notation sind hier ähnlich wie bei der Definition von Nicht-terminalen!

Die Eingabedatei fur einen Lexergenerator enthalt zum einen regulare Definitionen, und zum anderen Regeln, die einigen regularen Ausdrucken Aktionen zuordnen. Eine Aktion gibt an, was der zu erzeugende Lexer tun soll, wenn nextToken aufgerufen wird.

wietere Regeln:

1. Wenn der Lexer versucht, zu einem regularen Ausdruck eine passende Folge von Eingabezeichen zu finden, dann eine moglichst lange.

2. Es kann passieren, dass das gleiche maximal lange Prafix der Eingabe auf die regularen Ausdrucke mehrerer Lexerregeln passt. Jedenfalls bei vielen Lexergeneratoren wird in diesem Fall ein Lexer erzeugt, der immer die erste passende Regel benutzt.

6.3 Syntax-Analyse

Wir hatten im vorangegangenen Kapitel gesehen, wie man mit Hilfe nichtdeterministischer Kellerautomaten Syntaxanalyse fur kontextfreie Grammatiken durchfuhren kann. In der Praxis muss man die Aufgabe aber mit determinis-

tischen Algorithmen losen. Dazu werden entweder andere Algorithmen statt Kellerautomate nverwendet, oder man schränkt die kontextfreien Grammatiken so ein, dass man mit den deterministischen Kellerautomaten auskommt.

Es gibt zwei große Klassen kontextfreier Grammatiken, fur die Parser-Generatoren zu Verfugung stehen. Das eine sind die sogenannten LL(k)-Grammatiken (Top-Down), das andere die LR(k)-Grammatiken (Bottom-up). Das Ka steht für die anzahl vorausgesehner Zeichen im Speicher.

6.4 Der Algorithmus von Cocke, Younger und Kasami

6.4.1 Chomsky-Normalform fur kontextfreie Grammatiken

Eine kontextfreie Grammatik G(N, T, S, P) ist in Chomsky-Normalform, wenn sie den folgenden Einschrankungen genugt:

- Jede Produktion X $\rightarrow$ w mit X $\neq$ S hat als rechte Seite entweder ein Wort $w \in N^2$ (genau zwei Nichtterminalsymbole) oder ein Wort $w \in T$ (genau ein Terminalsymbol).
- Wenn es die Produktion S $\rightarrow$ E gibt, dann kommt S bei keiner Produktion auf der rechten Seite vor.

zu jeder Kontextfreien Grammatik gibt es eine äquivalente Kontextfreie Grammatik in Chomsky-Normalform.

Fur jede kontextfreie Grammatik G kann man die Menge EPS(G) = $X \in N | X \rightarrow^* E$ aller Nichtterminalsymbole berechnen, aus denen das leere Wort ableitbar ist. Vorgehen dabei:

1. Ermitteln, welche Nichtterminale in einem Schritt zu E führen
2. Ermitteln, welche Nichtterminale in einem Schritt zu der Menge aus 1 führen
3. ...

Zu jeder kontextfreien Grammatik G = (N, T, S, P) kann man eine kontextfreie Grammatik GN =(N, T, S, PN) berechnen, fur die gilt:

- L (GN) = L(G) ohne E und
- keine Produktion von GN hat E als rechte Seite.

Zu jeder kontextfreien Grammatik G = (N, T, S, P) kann man eine aquivalente kontextfreie Grammatik GN = (N, T, SN, PN) berechnen, fur die gilt:

- Wenn E nicht in L (G) ist, dann hat GN keine Produktion E als rechte Seite.
- Wenn E in L (G) ist, dann ist $S \rightarrow E$ die einzige Produktion von GN mit E als rechter Seite und SN kommt bei keiner Produktion auf der rechten Seite vor.

Vorgehen zur Überführung einer kontextfreien Grammatik in eine äquivalente kontextfreie Grammatik in der Chomsky-Normalform

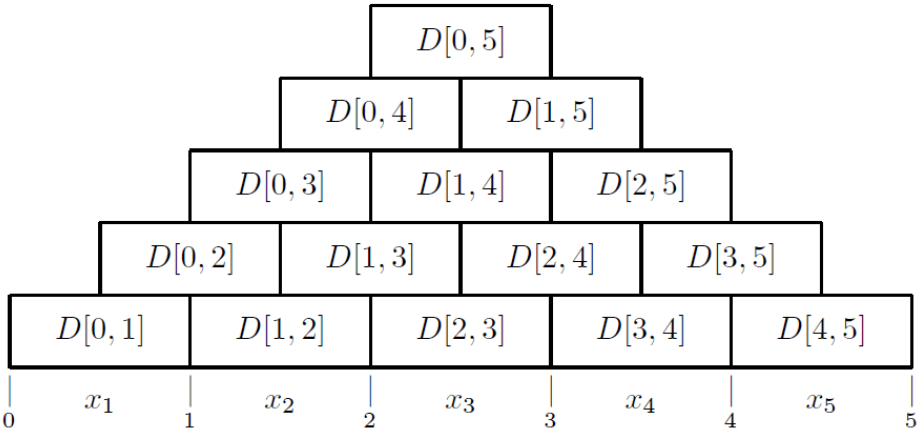
1. konstruiere aus G eine äquivalente Grammatik G_1, bei der die Terminalsymbole $a \in T$ nur in der Form $X \rightarrow a$ vorkommen.
2. erzeuge aus G_1 eine Grammatik G_2, für die die Eigenschaften der äquivalenten kontextfreien Grammatiken zutreffen.
3. erzeuge aus G_2 eine Grammatik G_3, die keine produktionen von NT auf NT enthält.
4. erzeuge aus G_3 eine Grammatik G_4, die keine Produktionen der Form $X \rightarrow Y_1 Y_2 ... Y_k, k >= 3$ hat.

Die Grammatik G4 enthält dann die gewünschten Eigenschaften.

6.4.2 Der Algorithmus von Cocke, Younger und Kasami

Der Algorithmus kümmert sich darum entscheiden zu können, ob ein Eingabewort zu einer bestimmten Sprache gehört. Dies kann CYK für jedes Wort aus Terminalsymbolen für eine Grammatik in Chomsky-Normalform auf deterministische Art und weise bestimmen. Dabei nehmen wir an, das jeder Buchstabe des Wortes in einer eigenen (Speicher-)Stelle liegt.

CYK betrachtet eine Datenstruktur die als Pyramide veranschaulicht werden kann. unter der Pyramide steht das zu beweisende Wort, in den einzelnen Zellen der Pyramide stehen Nicht-terminale, die den darunterliegenden Teil herleiten können. Ziel ist es, irgendwann das gesammte Wort auf das Startsymbol der Sprache zurückzuführen.



6.5 LR Parsing

Im folgenden wird immer eine Sprache für Erithmetische Ausrücke verwendet, die die NT E,T und F hat. Terminale sind *,+,(,) und a. Startsymol ist E, die Produktionen lauten:
$E \to E + T|T$
$T \to T * F|F$
$F \to (E)|a$
Im folgenden wird eine Ergänzung dieser Sprache behandelt, die ein neues Startsymbol E' mit einer einzigen Produktion für E' enthält.
$E' \to \dashv E \vdash$
$E \to E + T|T$
$T \to T * F|F$
$F \to (E)|a$
Eine Grammatik hat die Eigenschaft LR(K) für $k \in$, wenn durch den Kellerinhalt und die nächsten k Eingabezeichen der nächste Reduktionsschritt der Bottum-Up Syntaxanalyse eindeutig bestimmt ist. LR steht dabei dafür, dass die eingabe von Links kommt und die Rechtsableitung konstruiert wird. Dabei ist nur zu einem gegebenen k LR(k) gilt, nicht aber ob LR(k) für irgendein k gilt.

6.5.1 Prinzip der Syntaxanalyse für LR(K) Grammatiken

Ist eine Grammatik LR(k), so existiert eine Funktion act, die zu gegebenen Kellerinhalt und eingabe die nächste Aktion vorhersagen kann. Da die Menge der möglichen Eingaben unendlich ist, kann man die Daten nicht vorausberechnet in einer Tabelle schreiben. Statdessen werden Äquivalenzklassen für bestimmte Kellerinhalte gebildet, die auch zustände genannt werden. Den Zustand eines bestimmten Kellerinhaltes k beschreiben wir als z(k). Desweiteren sei die Funtion goto als goto(z(k),X) = z(kX). Existieren diese beiden Methoden act und goto, so kann man für eine LR(k)-Grammatik einen deterministischen Algorithmus für die bottum-up Syntaxanalyse angeben.

Automatenmodell:
Man erweitert den KA um einen zweiten Keller für die zustände, dieser ist zusammen mit dem ursprünglichen Keller immer gleich weit gefüllt. Für die Stelle i im Symbolkeller mit dem inhalt K gilt stets an der stelle i im Zustandskeller, dass z = z(K) gilt.

Algorithmus:
Im wesentlichen müssen die Fälle beachtet werden, dass ein weiteres Eingabesymbol gekellert wird (shift) oder das oberste Element Y des Symbolkellers mittels einer Produktion reduziert wird (reduce).

- **Shift:** ablegen des Eingabesymbols a im Symbolkeller und goto(z,a) im Zustandskeller, wobei z der zuletzt oberste Zustand des Zustandskellers ist.
- **Reduce:** In diesem Fall wird das obere Ende des Symbolkellers entfernt und durch A ersetzt. Im zsutandskeller werden ebensoviele elemente Entfernt und durch den Zsutand goto(z",A) ersetzt, Wenn z" der unter dem oberen Ende liegende Zustand ist.

Der Automat ist fertig, Wenn im Symbolkeller nurnoch der Startzustand steht.