Vorlesungsskript[1]

Finite-State-Methoden in der Sprachtechnologie

Institut für Computerlinguistik

Universität Zürich

http://www.cl.uzh.ch

Interaktive Lerneinheiten zur Vorlesung
http://kitt.cl.uzh.ch/kitt/clab/clabis/?vl=fsm

Simon Clematide
siclematide@cl.uzh.ch

Schriftliche Übungen:
Simon Clematide

Frühlingssemester 2013
Version von 29. April 2013

[1]PDF-Lauftext: http://files.ifi.uzh.ch/cl/siclemat/lehre/fs13/fsm/script/script.pdf
OLAT-Url: https://www.olat.uzh.ch/olat/url/RepositoryEntry/7302905864

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Organisatorisches

1.1 Infos

1.1.1 Unterlagen

Olat-Kurs und Lehrmaterialien
Campuskurs: "13FS 102/502 Finite-State-Methoden in der Sprachtechnologie"
`https://www.olat.uzh.ch/olat/url/RepositoryEntry/7302905864/CourseNode/77049798376275`

- *Folienskript* im 4-up-Format (farbig und s/w) als PDF-Dokument unter "Material"; 1-up Slides mit funktionierenden Links.
- *Lauftext* des Folienskripts mit Index (HTML und PDF) `http://files.ifi.uzh.ch/cl/siclemat/lehre/fs13/fsm/script/html/script.html`

Kursbücher

- K. R. Beesley und L. Karttunen (2003): Finite State Morphology, CSLI Publications. ISBN: 1-57586-434-7 `http://www.stanford.edu/~laurik/fsmbook/home.html` (Buch enthält Software, welche für Übungen intensiv benutzt wird). Ev. antiquarisch von Mitstudierenden erhältlich.
- Carstensen et al. (2009): Computerlinguistik und Sprachtechnologie: Eine Einführung. 2009.

1.1.2 Inhalt

Konzept und Inhalt der Vorlesung
Die VL "Finite-State-Methoden in der Sprachtechnologie" vermittelt

- die *formalen Grundlagen* der Endlichen-Automaten-Technik (EAT) (engl. *Finite State Methods* (FSM)).
- den *praktischen Einsatz* der *Xerox Finite State Tools* (*XFST* bzw. FOMA), um die wichtigsten Anwendungen der FSM in der Sprachtechnologie programmieren zu können: morphologische Analyse und Generierung, Tokenisierung, Spelling Correction, Named Entity Recognition, Chunking, flache syntaktische Analyse
- die *grundlegende Theorie* zur Beschreibung von morphologischen Erscheinungen in verschiedenen natürlichen Sprachen.

1.1.3 Lernform

Projektorientierte Vorlesung

- Ziel: Selbständige Durchführung eines kleinen *Schluss-Projekts* mit Finite-State-Methoden aus dem Bereich: Named-Entity-Recognition im biomedizinischen Bereich, Morphologiesystem für Rätoromanisch weiterentwickeln, Adaption von einem Morphologiesystem für Deutsch für ältere Sprachstufen oder eigene Ideen!
- Lehrform: Theoretische Inputstunden (ca. 11 Termine; normalerweise 14-15.45h im BIN-1-D-7) mit anschliessendem praktischen Arbeiten im Computerraum (normalerweise 16-17.30h im BIN-0.B.04) mit persönlicher Betreuung!
- Übungsstunde: Teilpräsenz ist obligatorisch für Besprechung der Übungen, sowie Planung und Besprechungen für das Schlussprojekt
- Aufwand für Vorlesung: 4 ECTS-Punkte, d.h. 120h Arbeit

1.1.4 Leistungsnachweis

Schriftliche Übungen (SU), Schluss-Projekt und schriftliche Prüfung

- *5 Übungseinheiten* zur Abgabe ab Woche 3
- 25% der Schlussnote
- Benotung (in Zehntel-Schritten): 6 (4-5 SU); 4.6 (3 SU); 3.3 (2 SU); 2 (1 SU); 1 (0 SU)
- SU kann zu 0/0.5/1 bestanden sein
- Teilweise mit Musterlösungen, aber auch Fragen und Diskussion in der Übungsstunde
- *Schluss-Projekt*: 25% der Schlussnote
- *Schriftliche Schlussprüfung*: 50% der Schlussnote

Schriftliche Schluss-Prüfung

- Zeit: Montag, 3.6.2013 14-15.45h
- Mögliche Form: Reine Theorieprüfung oder mit praktischem Teil an Computer
- Stoff: Skript, Übungen, Pflichtlektüre
- Bitte das für Sie gültige Infoblatt zur Leistungsüberprüfung genau lesen! [ICL 2011a, ICL 2011b]

1.2 Software

Software für Übungen

- Die Arbeit mit den XFST-Werkzeugen kann remote via `ssh` und X11 auf unserm Server `kitt.cl.uzh.ch` erledigt werden.
- XFST-Werkzeuge stehen für Plattformen MacOS X, Windows und Linux zur Verfügung

- Mit dem FOMA-Werkzeug (`https://code.google.com/p/foma/`) steht eine open-source-Reimplementation zur Verfügung, welche den Befehlsumfang von xfst unterstützt, aber weniger hilfreich beim Debuggen ist.
- Auf allen Macs im Arbeitsraum BIN-0.B.04 sind XFST und FOMA lokal installiert.

Kapitel 2

Was ist Morphologie?

Herzlichen Dank an Manfred Klenner für Quelltexte.

Lernziele

- Kenntnis der grundlegenden Thematik von Morphologie: Flexion und Wortbildung
- Kenntnis der traditionellen morphologischen Begriffe Deklination, Konjugation, Paradigma, Komposition, Derivation, Neoklassische Wortbildung, Konversion, Kürzung
- Kenntnis des strukturellen Aufbaus von Wörtern, insbesondere von deutschen Komposita
- Fähigkeit zur Wortbildungsanalyse von deutschen Wörtern
- Kenntnis über Themen und Anwendungen der Computermorphologie

2.1 Die Disziplin "Morphologie"

Der Begriff *Morphologie* in der Linguistik

Definition 2.1.1 (Traditionell)**.** *Morphologie* ist die Lehre von den Wortformen und ihrem Innenbau.

Entstehung
1859 übertrug der deutsche Sprachwissenschaftler A. Schleicher den Begriff auf die Linguistik:

> «Für die Lehre von der Wortform wähle ich das Wort „Morphologie".»

A. Schleicher führte auch den Begriff „Linguistik" ein, übertrug Darwins Evolutionstheorie auf die Entwicklung der Sprache und war wesentlich an der Rekonstruktion des Indoeuropäischen beteiligt.

Traditionelle Sprachlehre
Aufteilung in Laut-, Wort- und Satzlehre.

Abbildung 2.1: August Schleicher (1821–1868)

Traditionelle Einteilung der Morphologie

Flexion: Lehre von den syntaktischen Wortformen

- *Konjugation* von Verben
- *Deklination* von Nomen, Adjektiven, Artikeln und Pronomen
- *Steigerung* von Adjektiven (und wenigen Adverbien)

Wortbildung: Lehre von der Schöpfung und lexikalischen Verwandtschaft von Wörtern

- *Derivation* (Ableitung): Bild → bildlich
- *Komposition* (Zusammensetzung): Wort, Bildung → Wortbildung
- *Konversion* (Null-Ableitung, Umkategorisierung): bilden → das Bilden

2.2 Flexion

2.2.1 Der Begriff "Stamm"

Flexion = Stamm + Endung

Definition 2.2.1. *Flexion* bezeichnet die Bildung von unterschiedlichen syntaktischen Wortformen aus einem (Wort-)Stamm.

Flexionsendungen (Flexionssuffix)
In vielen (europäischen) Sprachen entstehen die Wortformen, indem Endungen an einen Stamm angehängt werden.

Definition 2.2.2 (Flexionsstamm (engl. *stem*))**.** Ein *Wortstamm*, kurz Stamm, ist der gemeinsame lexikalische Kern von syntaktischen Wortformen, welche um Flexionsendungen (selten Präund/oder Infix) gekürzt sind.

	Singular			Plural		
	Artikel	Stamm	Endung	Artikel	Stamm	Endung
Nominativ	das	Kind	-	die	Kind	er
Akkusativ	das		-	die		er
Dativ	dem		- e	den		ern
Genitiv	des		es s	der		er

Quelle: `http://www.canoo.net`

Abbildung 2.2: Wortform = Stamm + Endung

Beispiele: Deutsche Stämme

```
Legende: <-...-> = Stamm, +...+ = Endung

Unwägbarkeiten      übermenschliches
<----------->++     <------------>++
verliebst           schönstem
<----->++           <--->
ging                gegangen
<-->                  <-->
die                 Häuser
?                   <---->
brechenden          sähest
<--->               <->
<------>
rote                bin
<->                 ?
```

Hinweis: Ob die Steigerung von Adjektiven tatsächlich als Flexionsphänomen betrachtet werden soll, ist durchaus diskussionswürdig.

Beispiele: Stamm-Endung-Analysen

	Singular			Plural		
	Artikel	Stamm	Endung	Artikel	Stamm	Endung
Nominativ	das	Haus	-	die	Häus	er
Akkusativ	das		-	die		er
Dativ	dem		- e	den		ern
Genitiv	des		es	der		er

Quelle: `http://www.canoo.net`

Abbildung 2.3: Nominal-Flexionsparadigmen mit mehr als einem Stamm

Präsens							
Indikativ				Konjunktiv I			
Person	Stamm	ZM	Endung	Person	Stamm	ZM	Endung
ich	bin			ich			-
du	bist			du		e	st
er/sie/es	ist		-	er/sie/es	sei		-
wir	sind			wir			n
ihr	seid			ihr		e	t
sie	sind			sie			n

Quelle: `http://www.canoo.net`

Abbildung 2.4: Ausschnitt aus Flexionsparadigma mit mehr als einem Stamm

2.2.2 Flexionsmerkmal

Flexionsmerkmale und Flexionswerte

Definition 2.2.3 (auch Flexionskategorie)**.** Ein *Flexionsmerkmal* ist eine grammatische Kategorie, welche die Flexion in den syntaktischen Wortformen mit unterschiedlichen Werten ausgeprägt.

Definition 2.2.4 (auch morphosyntaktischer Merkmalwert)**.** *Flexionsmerkmalswerte* sind sprach- und wortartenabhängige Ausprägungen von grammatischen Kategorien durch die Flexion.

Beispiel 2.2.5 (Flexionsmerkmal Kasus). Das Flexionsmerkmal Kasus kann in vielen Sprachen mit den Werten Nominativ und Akkusativ erscheinen.

Beispiel 2.2.6 (Kasus im Deutschen). Im Standarddeutschen können Substantive bezüglich Kasus mit den Flexionsmerkmalswerten Nominativ, Akkusativ, Dativ und Genitiv ausgeprägt sein.

Beispiel: Kasus im Ungarischen

Kasus	Wortform	Bedeutung
Nominativ	ház	das Haus
Dativ	háznak	dem Haus
Akkusativ	házat	das Haus
Instrumental-Komitativ	házzal	mit dem Haus
Kausal-Final	házért	wegen des Hauses
Translativ	házzá	(wird) zum / zu einem Haus
Terminativ	házig	bis zum Haus

Tabelle 2.1: Kasussystem im Ungarischen

2.2.3 Flexionsparadigma

Definition 2.2.7 (Flexionsmuster)**.** Ein *Paradigma* ist eine Menge von syntaktischen Wortformen, welche zusammen ein Deklinations- oder Konjugationsmuster bilden.

Beispiel 2.2.8 (Substantivparadigma).

Frage: Wieviele Einträge umfasst ein Paradigma eines Adjektivs?

Wortform	Kasus	Numerus	Genus
Gedanke	Nominativ	Singular	Maskulinum
Gedanken	Genitiv		
Gedanken	Akkusativ		
Gedanken	Dativ		
Gedanken	Nominativ	Plural	
Gedanken	Genitiv		
Gedanken	Akkusativ		
Gedanken	Dativ		

Exhaustive vs. distinktive Analyse

Exhaustive Analyse nach [HAUSSER 2001, 246]
Jede mögliche Kombination von Flexionsmerkmalswerten wird als einzelne Analyse zurückgegeben, auch wenn die Wortformen sich nicht unterscheiden.

Distinktive Analyse nach [HAUSSER 2001, 246]
Es werden nur solche Analysen zurückgeliefert, wo sich die Wortformen (grundsätzlich) unterscheiden. Deutsche Adjektive nur 18 verschiedene Formen.

Beispiel 2.2.9 (Grade von Exhaustivität bei Analysen von „schnelle“).

- GERTWOL: `http://www2.lingsoft.fi/cgi-bin/gertwol`
- TAGH: `http://www.tagh.de`

2.3 Wortbildung

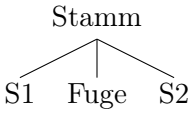
2.3.1 Komposition

Wortzusammensetzung (Komposition)

Definition 2.3.1 (Komposition (engl. *compounding*))**.** Bei der *Komposition* wird ein Erstglied (Stamm, ev. Wortform) mit einem zweiten Stamm (Zweitglied) zu einem neuen Stamm zusammensetzt. Vom Erstglied abhängig kann ein Fugenelement dazwischen treten.
Das *Zweitglied* bestimmt immer die Wortart und die Flexionsklasse.

Beispiel 2.3.2 (Nominal-Komposita).

- N → N N : Teppichboden, Bücherregal, Arbeitszeugnis
- N → Adj N : Tiefgang, Rotstift
- N → V N : Schreibpapier, Tippfehler
- N → Adv N : Abwärtstrend, Aussensensor
- N → P N : Abbitte, Vorwort
- N → Pron N : Wir-Gefühl, Ich-Mensch
- N → Numerale N : Erststimme, Achterbahn



Komposition im Deutschen

- Adj → Adj Adj : frühreif
- Adj → V Adj : fahrbereit
- Adj → Pron Adj : selbstherrlich
- Adv → Adv Adv : dorthin, hierher
- Adv → P Adv : zwischendrin, hinterher
- V → N V : teilnehmen, gewährleisten
- V → Adj V : trockenlegen, wachrütteln
- V → V V : gefriertrocknen, presspolieren
- V → Adv V : fortschicken

Worum handelt es sich bei "Taugenichts", "Nimmersatt" oder "«Ich bin Borat»-Kit"?

Fugenelemente
(a) wie Nominativ-Plural-Suffix (paradigmisch)

Schwester Uhr	Paar Kasten	Schwester-n-paar Uhr-en-kasten

(b) wie Genitiv-Singular-Suffix (paradigmisch)

Bauer Jahr	Frau Zeit	Bauer-s-frau Jahr-es-zeit

(c) Fugen-s (bei Feminina) (unparadigmisch)

Arbeit Liebe	Anzug Brief	Arbeit-s-anzug Liebe-s-brief

(d) Tilgung von Schwa im Auslaut des Erstgliedes

Schule	Buch	Schul-buch

Konstituentenstruktur von rekursiven Komposita

Wortgrammatik und Betonungsverhältnis
Der Kopf, d.h. das morpho-syntaktisch und semantisch bestimmende Element, steht im Deutschen immer als Zweitglied. Allerdings trägt gerade das Erstglied den Wortakzent: "FRAUENfilmfest" vs. "FrauenFILMfest". Abweichungen vom Normalbetonungsmuster wird als Kontrastbetonung interpretiert.

Parsebäume für Morphologieanalyse
Wie sieht eine Konstituentenanalyse für folgende Wörter aus?
„Kohlenstoffkreislauf"
„Krankenkassenkostendämpfungsgesetz"

Beispiel 2.3.3 (Hörbeispiel für Betonungsmuster in rekursiven Komposita).
[Morgan und Stocker 2007]

Semantische Klassifikation [BUSSMANN 2002]

- *Determinativkomposita* (sehr produktiv) → Nichtkopf bestimmt den Kopf näher; in welche Richtung die Bedeutung spezialisiert wird, ist weitgehend offen: "Bürogebäude", "Zigarettenraucher"
- *Kopulativkomposita* → Glieder sind gleichrangig, koordinierte Beziehung zwischen Denotaten, beziehen sich auf gleiche Entität, gleiche Wortklasse: "Dichter-Fürst", "süss-sauer"
- *Possessivkomposita* → Das Kompositum bezeichnet (oft pars pro toto) prominente Eigenschaft des Gemeinten: "Trotzkopf", "Langfinger", "Milchgesicht"

2.3.2 Derivation

Wortableitung (Derivation)

Definition 2.3.4 (Derivation)**.** Die *Derivation* ist ein Wortbildungsvorgang, bei dem normalerweise ein Stamm mit einem Derivationsaffix (Präfix, Suffix) verbunden wird.

Definition 2.3.5 (Derivationsaffix)**.** *Derivationsaffixe* kommen nicht als selbständige Wörter vor. *Derivationssuffixe* können Wortartenwechsel auslösen, d.h. sie tragen Wortartinformation.

Innere Derivation
Derivation, bei der eine Lautveränderungen im Innern eines Stammes geschieht, wird manchmal 'innere Derivation' genannt: "werfen" → "Wurf", "fallen" → "fällen".

Beispiele Derivation

Beispiel 2.3.6 (Suffigierung).
N → A : wissentlich, heilig
V → A : ausführbar, erklärlich, folgsam
A → N : Reichtum, Freiheit, Dichte
N → N : Menschheit, Freundschaft, Briefchen, Köchin
V → N : Lehrer
A → V : bleichen, faulen, steinigen
N → V : gärtnern, knechten, schneidern, buchstabieren
V → V : säuseln, hüsteln

Beispiel 2.3.7 (Präfigierung).
N : Unschuld, Misserfolg
V : entziehen, versprechen (sehr häufig)
A : unschuldig

2.3.3 Weitere Wortbildungsphänomene

Neoklassische Wortbildung
Die *Neoklassische Wortbildung* ist ein produktiver Wortbildungsvorgang, bei dem ein lateinisches oder griechisches Wortbildungselement (Formativ, auch Konfix genannt) mit weiteren Formativen oder Derivationsaffixen verbunden wird.
Die Fugenelemente "o" oder "i" können auftreten.

Beispiel 2.3.8.

- Psychologie, Psychiatrie, psychosozial, psychisch
- Psychologie, Logopäde, logisch
- Psychopath, Pathologe, pathogen

Frage
Aus welchen Bestandteilen besteht das Wort "Morphologie"?

Der Duden-Grammatik [Dudenredaktion 2005, §994] nennt neoklassische Wortbildungselemente "Konfix". Daraus ergeben sich dann Konfixkomposita und Konfixderivate.

Konversion, Kürzung, Zusammenrückung

Definition 2.3.9 (Konversion)**.** Die *Konversion* ist ein Wortbildungsvorgang, bei dem Wortartenwechsel ohne Affigierung statt findet: "Frühstück" → "frühstücken".

Definition 2.3.10 (Zusammenrückung)**.** Die *Zusammenrückung* ist ein Wortbildungsvorgang, bei dem syntaktische Fügungen substantiviert werden: "Vergissmeinnicht", "Taugenichts", "«Ich bin Borat»-Kit".

Definition 2.3.11 (Kürzung (engl. *clipping*))**.** Die *Kürzung* ist ein Wortbildungsvorgang, bei dem Wortanfänge ("Omnibus" → "Bus"), Wortenden ("Universität" → "Uni") oder Wortmitten ("Fernsprechamt" → "Fernamt") ausgelassen werden.

2.3.4 Der Begriff „Wurzel"

Definition 2.3.12 (engl. *root*, auch Kern)**.** Die *Wurzel* bezeichnet einen oder mehrere Teile einer Wortform, welcher nach Entfernung aller Flexions- und Derivationsbestandteile mit den Mitteln der Komposition nicht weiter analysiert werden kann.

Beispiel 2.3.13 (Wurzeln in deutschen Wörtern).

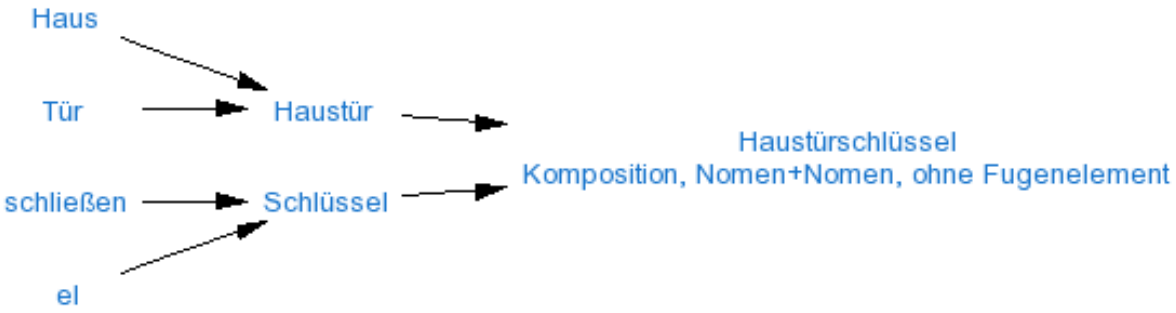
```
Beispiele (<-...-> = Wurzel):

Unwägbarkeiten    übermenschliches   veruntreust
  <->                 <---->              <-->
ärmsten           Fensterkreuz       Haustürschloss
<->               <-----><--->       <--><-><----->
konnten           vergräbst          Weisheit
<-->                 <-->            <-->
```

Hinweis: Wörter können mehr als eine Wurzel enthalten. Wurzeln müssen keinen existierenden Wörtern entsprechen.

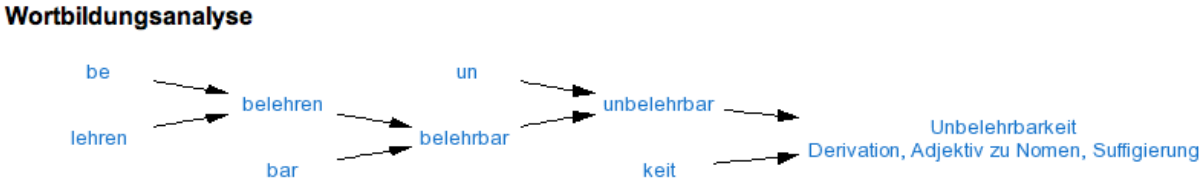
Beispiele: Wortbildungsanalysen

2.4 Computermorphologie

2.4.1 Anwendungen



Quelle: `http://www.canoo.net`



Quelle: `http://www.canoo.net`

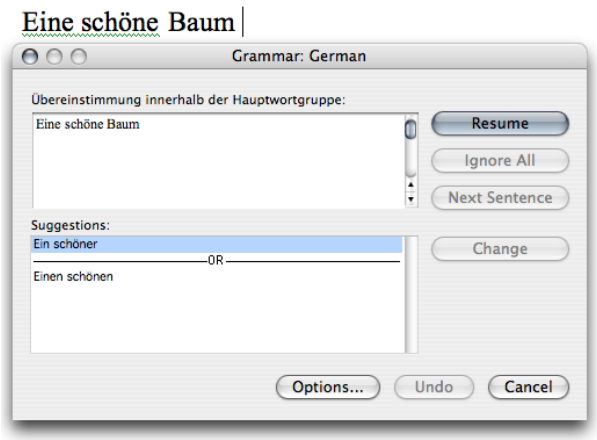


Abbildung 2.5: Grammatik-Korrektur von Kongruenzfehlern in Microsoft Word

Beispiel: Grammatik-Korrektur in Microsoft Word 2004
Siehe Abb. 2.5 auf Seite 17. Welches linguistische Wissen wird hier eingesetzt?

Anwendungen der Computermorphologie

- *Grammatikkorrektur*: Sind morphosyntaktische Kongruenzen innerhalb einer NP erfüllt?
- *Silbentrennung*: Wie trennt man "erklecklich"?
- *Text-To-Speech*-Systeme (Graphem-zu-Phonem-Konversion): Wie spricht man "th" in "hothouse" aus?
- *Rechtschreibe-Korrektur*: Wie unterscheidet man zwischen falsch geschriebenen und dem Korrektursystem unbekannten Wörtern?

- *Tokenisierung*: Wie erkennt man in Schriftsystemen wie dem Chinesischen die lexikalischen Einheiten (Wörter), welche aus einem oder mehreren Graphemen (Silbenschrift) zusammengesetzt sind?
- *Information Retrieval, Text Mining*: Wie kann in morphologisch reichen Sprachen die Grundform von Wörtern zur Indexierung berechnet werden?
- *Information Extraction*: Wie kann ich die flektierten Formen eines Lemmas finden?
- *Parsen*: Welche morphosyntaktischen Merkmale kann eine flektierte Wortform realisieren? Wie finde ich Valenz- oder semantische Information in einem lemma-basierten Lexikon?
- *Generierung*: Welche flektierte Wortform drückt die verlangten morphosyntaktischen Merkmale aus?

2.4.2 Typische Berechnungen

Lemmatisierung und Morphologieanalyse

Definition 2.4.1 (Grundformbestimmung)**.** Die *Lemmatisierung* ist die Bestimmung der Lemmas, welche einer Wortform zugrunde liegen.

Definition 2.4.2 (Morphologieanalyse)**.** Die *Morphologieanalyse* ist die Bestimmung der morphologischen Merkmale einer Wortform bezogen auf ein bestimmtes Lexem.

Lemmatisierung und Morphologieanalyse mit GERTWOL

http://www2.lingsoft.fi/cgi-bin/gertwol

```
Verbrechens
 "Verb#rechen"  S MASK SG GEN
 "Verb#rech~en"  S NEUTR SG GEN
 "Ver|brech~en"  S NEUTR SG GEN

eine
 "ein" ART INDEF SG NOM FEM
 "ein" ART INDEF SG AKK FEM
 "einer" PRON INDEF SG NOM FEM
 "einer" PRON INDEF SG AKK FEM
 "ein~en" V IND PRÄS SG1
 "ein~en" V KONJ PRÄS SG1
 "ein~en" V KONJ PRÄS SG3
 "ein~en" V IMP PRÄS SG2
```

Typische Berechnungen in der Computermorphologie

Frage
Wie heissen die Fachbegriffe für die folgenden Berechnungsprobleme?

- Ist die Zeichenkette x eine gültige Wortform der Sprache L?
- Welche Wortart soll der Wortform x im Kontext C zugewiesen werden?
- Welche morphosyntaktischen Merkmale drückt die Wortform x aus?

- Welche möglichen Lemmata liegen der Wortform x zugrunde?
- Welche Wortformen realisieren das Lemma l mit den morphologischen Merkmalen m_1 bis m_n?
- Aus welchen morphologischen Bestandteilen besteht eine Wortform x?
- Wie sieht die morphologische Konstituentenstruktur einer Wortform x aus?
- ...

2.5 Vertiefung

- Linguistisches Propädeutikum I: Morphologie aus ECL I, insbesondere für die Begriffe Token, Wort, Lemma, Lexem und morphologische Kategorie: `http://kitt.cl.uzh.ch/clab/lingProp`
- Abschnitt Morphologie in [Carstensen et al. 2009]

Übung

Was haben wir gelernt?

Verlust. Der russische Ölmilliardär **Roman Abramowitsch** hat sich von seiner Ehefrau Irina und damit von einem erklecklichen Teil seines 23-Milliarden-Franken-Vermögens getrennt. Satte 13 Milliarden

soll die Ex erhalten haben, dazu ein paar nette Accessoires wie Villa, Luxusjacht oder Privatjet. Da bekommt das Wort Schmerzensgeld eine völlig neue Bedeutung.

Abbildung 2.6: Wortbildung im Migros-Magazin vom 19.3.2007 Seite 4 `http://www.migrosmagazin.ch`

Welche morphologischen Eigenheiten weist dieser Text auf? Versuchen Sie die Terminologie anzuwenden.

Kapitel 3

Sanfter Einstieg in FSM

Herzlichen Dank an K. Beesley für das Zurverfügungstellen der Folien, welche von `http://www.stanford.edu/~laurik/fsmbook/lecture-notes/Beesley2004/index.html` stammen (mit Quelle B04 versehen im Folgenden)

Lernziele

- Informeller Einstieg in die zentralen Ideen hinter Endlichen Automaten (Finite State Automaton), regulären Sprachen und Relationen
- Übersicht über Hauptanwendungsgebiete der Finite-State-Methods (FSM) in der Sprachtechnologie
- Kennenlernen der graphischen Darstellung von endlichen Netzwerken (Automaten/Transduktoren)
- Einsicht in die Formalisierung von Sprachen als Mengen
- Anwendung von Mengenoperationen auf Sprachen
- Verstehen des Konzepts der Konkatenation von Sprachen und Netzwerken

3.1 Motivation

Langfristige FSM-Kursziele

Verständnis für

- reguläre (auch rationale) Sprachen und Relationen (Paar von Sprachen)
- endliche Netzwerke (endliche Automaten und Transduktoren)
- mathematische Operationen über Sprachen bzw. Netzwerken
- die Beziehung zwischen Sprachen/Relationen, Netzwerken und regulären Ausdrücken

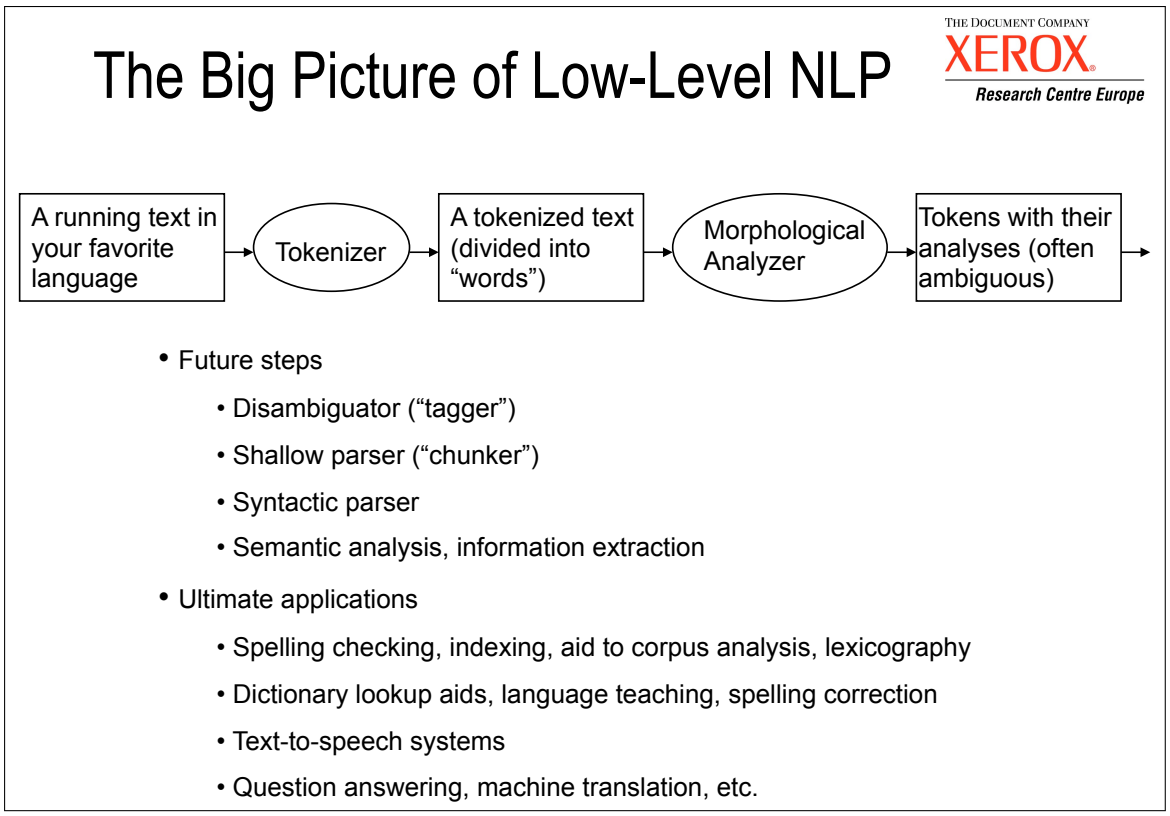
Fähigkeiten

- Netzwerke mit xfst-Technologie zu programmieren, welche nicht-triviale NLP-Applikationen ermöglichen
- Methodenkompetenz: Was ist einfach, möglich, schwierig mit FSM?

Warum Finite-State-Methoden verwenden?

Finite-State-Systeme sind

- mathematische elegant, flexibel und leicht modifizierbar.
- effizient (Rechenzeit) und kompakt (Speicherverbrauch) für typische NLP-Tasks.
- inhärent bidirektional. D.h. sie können analysieren und generieren.
- sprachunabhängig geeignet für „lower-level"-NLP: Tokenisierung, Rechtschreibekorrektur, Phonologie, Morphologie, Tagging, NER, Chunking, flache Syntaxanalyse



Quelle: B04

Warum Finite-State-Methoden verwenden?

Deklarativität und sprachunabhängiger Formalismus

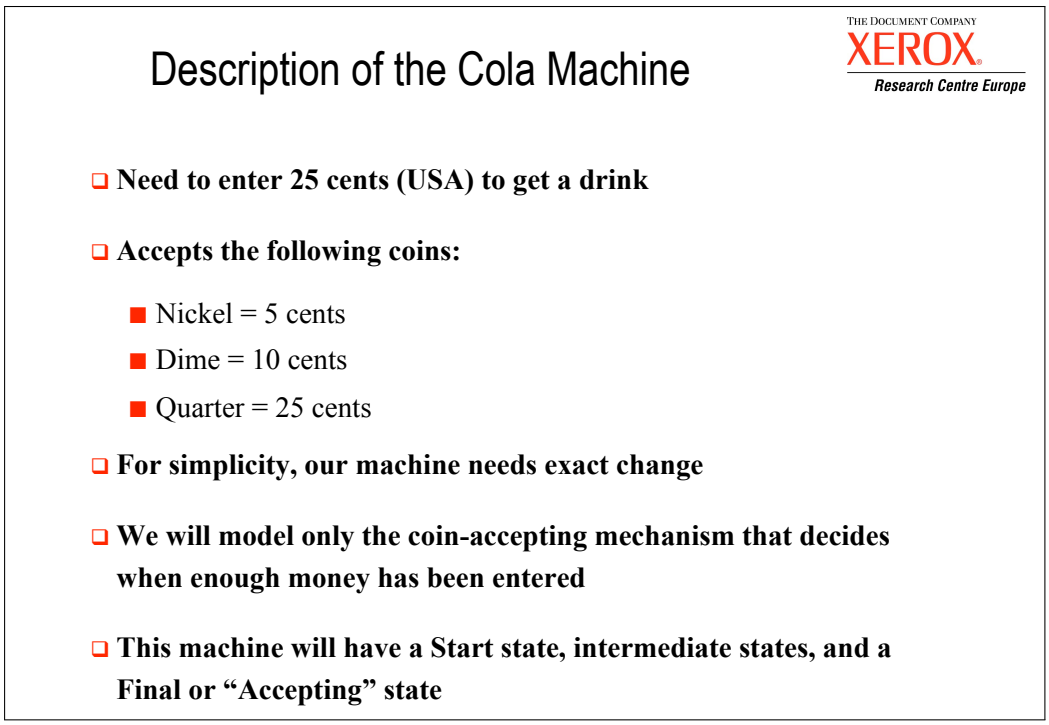
- Deklarative Programmierung: Wir beschreiben die Fakten von natürlicher Sprache wie Linguisten und schreiben keine Prozeduren.
- Der Anwendungskode ist in Laufzeitapplikationen (oder API-Aufrufen) gekapselt und absolut sprachunabhängig.

„Finite-State Mindset" entwickeln

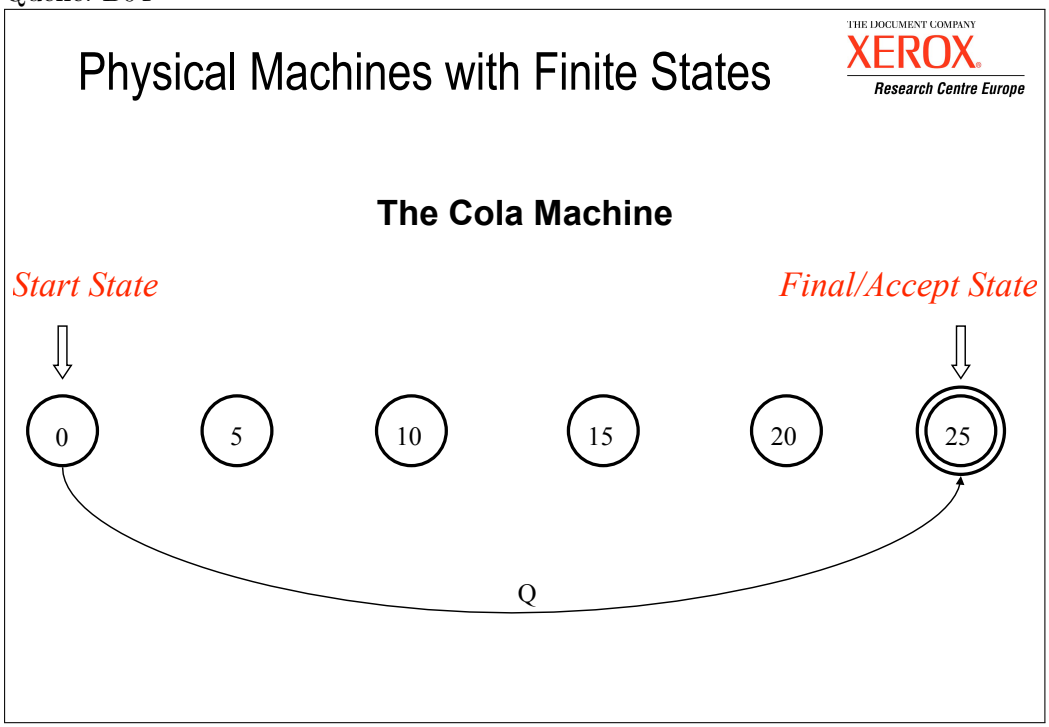
Das Lernen der Notationen, der FSM-typischen Dekompositionen der Probleme, der typischen Programmieridiome und -tricks braucht etwas Zeit und Übung.

3.2 Endliche Automaten (EA)

3.2.1 Physische EA



Quelle: B04



Quelle: B04 (modifiziert von SC)

The Cola Machine Language

THE DOCUMENT COMPANY XEROX Research Centre Europe

- **List of all the sequences of coins accepted by the Cola Machine**
  - Q
  - DDN
  - DND
  - NDD
  - DNNN
  - NDNN
  - NNDN
  - NNND
  - NNNNN

And Here is Where We Make the Step from Physical Machines to Linguistic Machines:

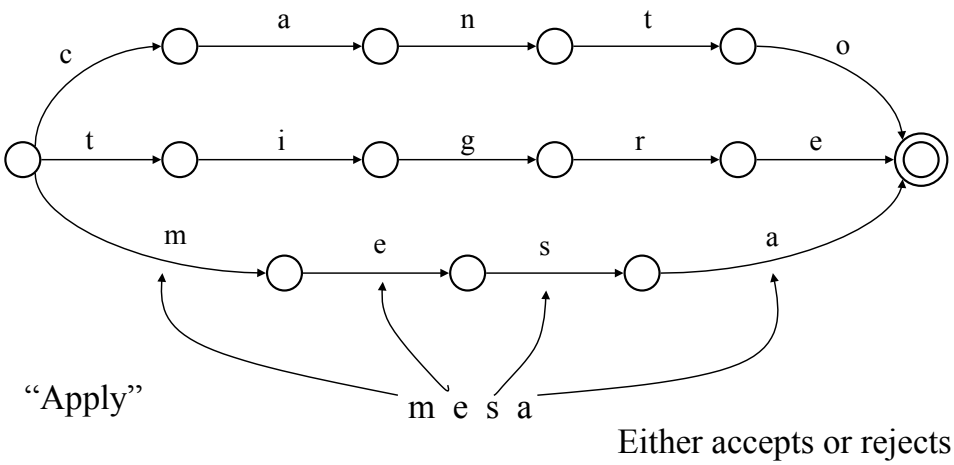
- Think of the coins as SYMBOLS or CHARACTERS
- The set of symbols accepted is the ALPHABET of the machine
- Think of accepted sequences of coins as WORDS or "strings"
- The set of words accepted by the machine is its LANGUAGE

Quelle: B04

3.2.2 Linguistische EA

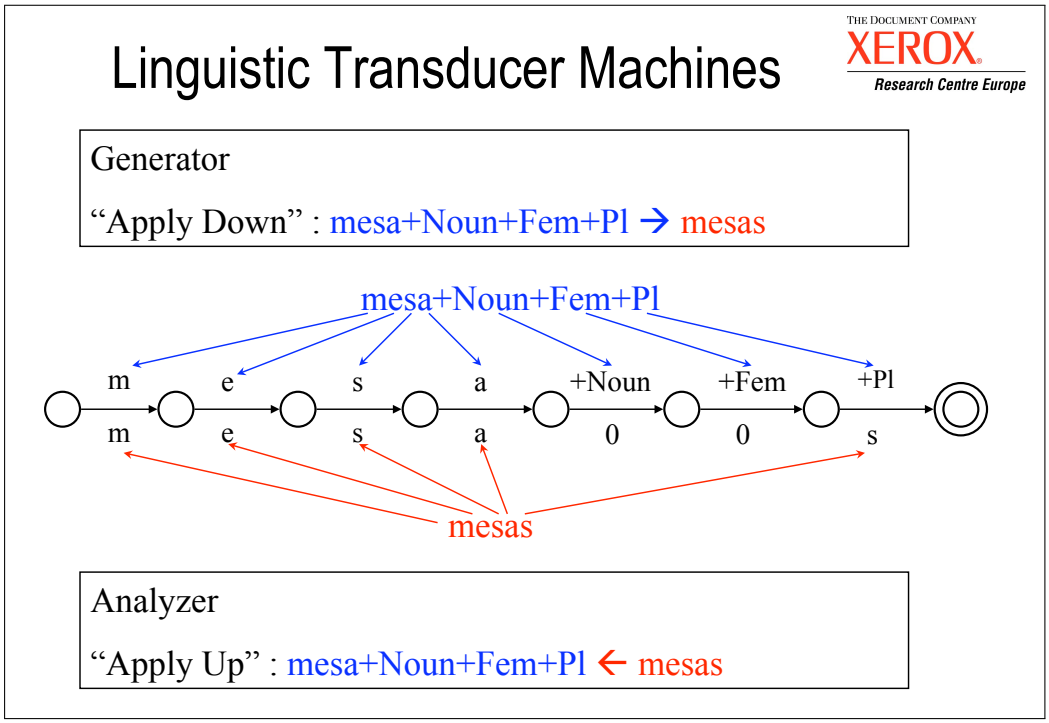
Linguistic Machines

THE DOCUMENT COMPANY XEROX Research Centre Europe



Quelle: B04

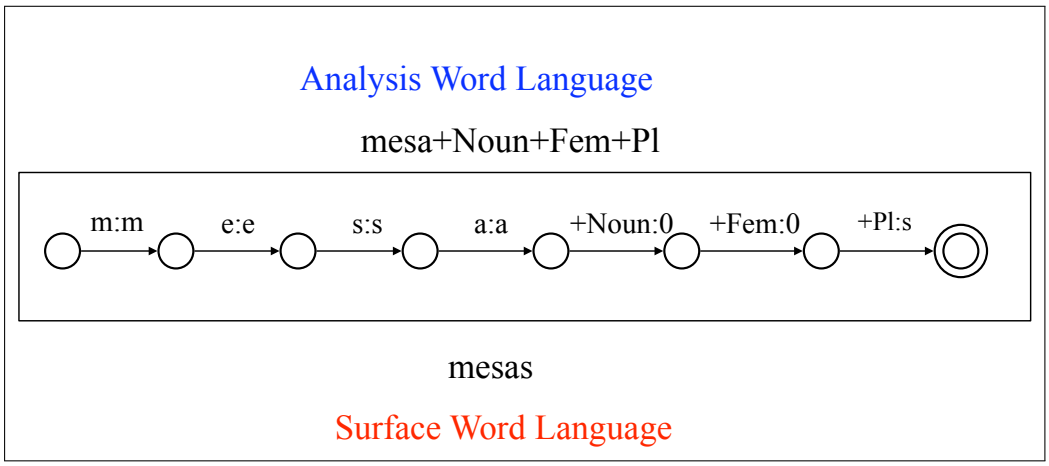
3.2.3 Endliche Transduktoren (ET)



Quelle: B04 (modifiziert von SC)

Transduktoren und ihre 2 Sprachen

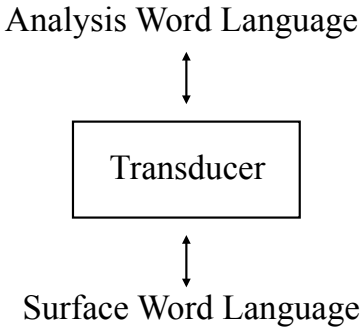
Notationskonvention
Symbole aus oberer Sprache werden normalerweise links und Symbole aus untere Sprache rechts von Doppelpunkt notiert.



Quelle: B04

A Morphological Analyzer

THE DOCUMENT COMPANY XEROX Research Centre Europe



Quelle: B04

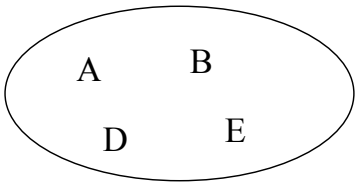
3.3 Sprachen als Mengen

3.3.1 Mengen

A Quick Review of Set Theory

THE DOCUMENT COMPANY XEROX Research Centre Europe

A set is a collection of objects.



We can enumerate the "members" or "elements" of finite sets: { A, D, B, E }.

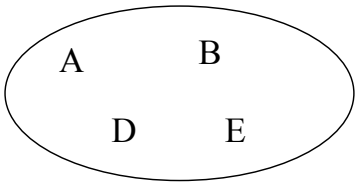
There is no significant order in a set, so { A, D, B, E } is the same set as { E, A, D, B }, etc.

Quelle: B04

Uniqueness of Elements

THE DOCUMENT COMPANY XEROX Research Centre Europe

You cannot have two or more 'A' elements in the same set

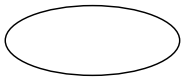
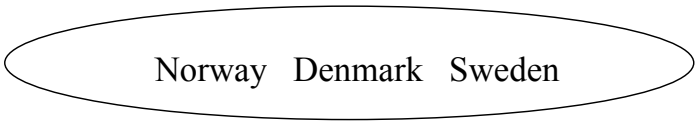


{ A, A, D, B, E } is just a redundant specification of the set { A, D, B, E }.

Quelle: B04

Cardinality of Sets

THE DOCUMENT COMPANY XEROX Research Centre Europe

- **The Empty Set:** 
- **A Finite Set:** 

- **An Infinite Set: e.g. The Set of all Positive Integers**

Quelle: B04

3.3.2 Formale Sprachen

Formal Languages

THE DOCUMENT COMPANY XEROX Research Centre Europe

Very Important Concept in Formal Language Theory:

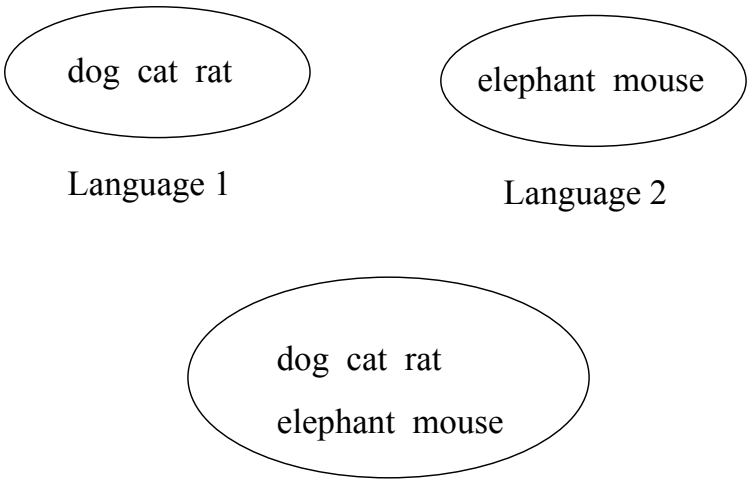
A *Language* is just a Set of Words.

- We use the terms "word" and "string" interchangeably.
- A Language can be empty, have finite cardinality, or be infinite in size.
- You can union, intersect and subtract languages, just like any other sets.

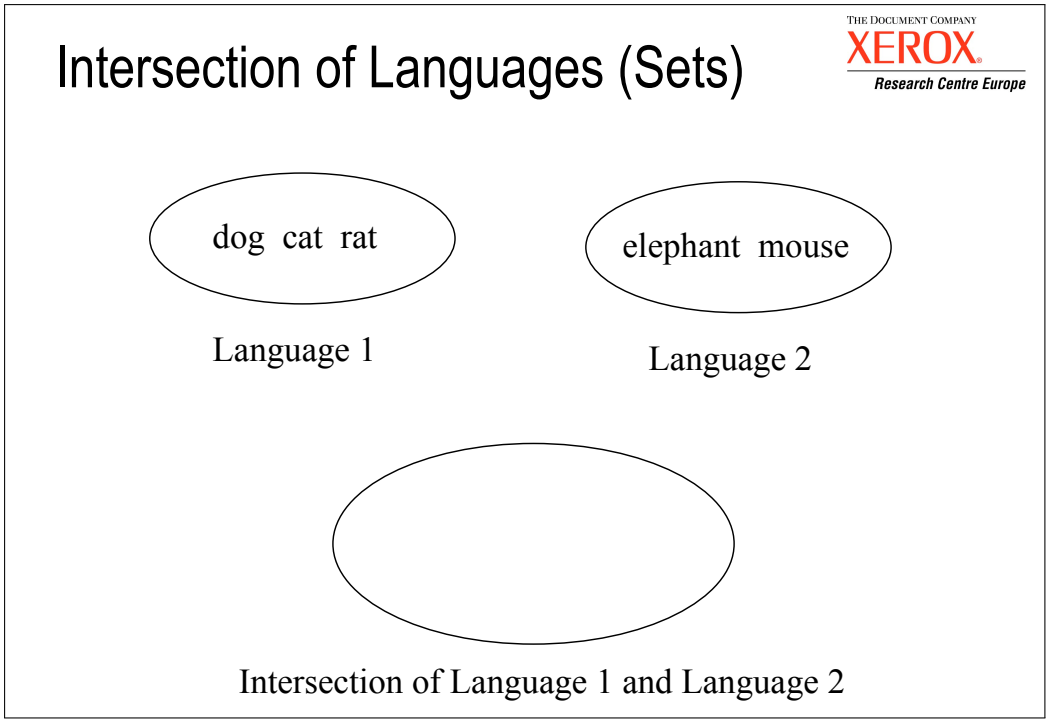
Quelle: B04

Union of Languages (Sets)

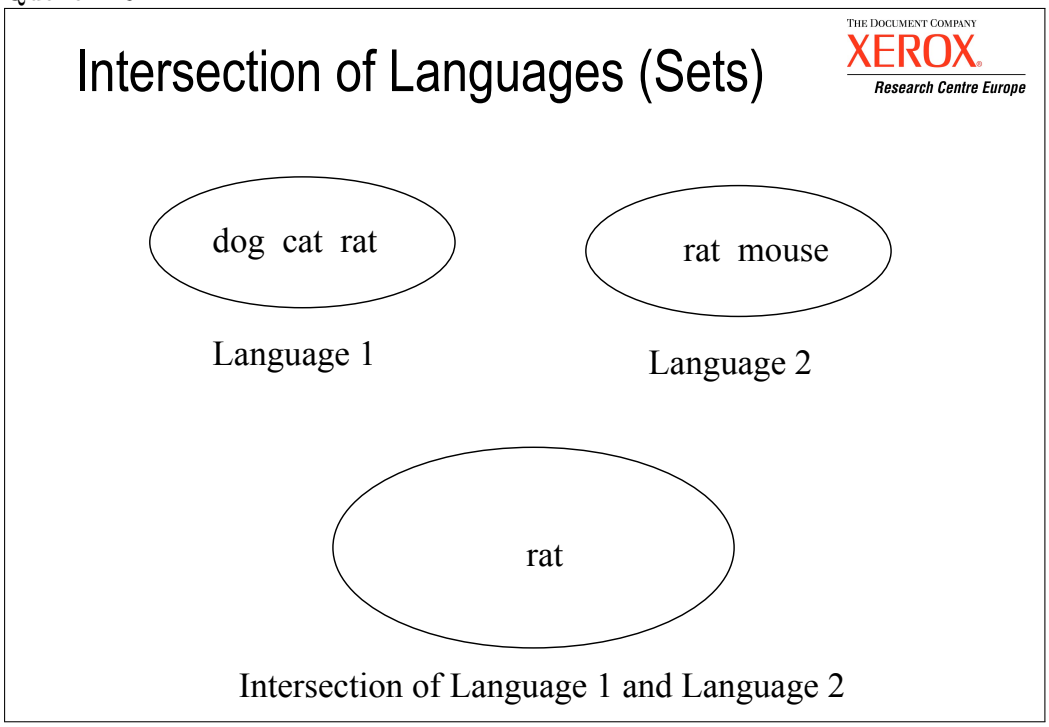
THE DOCUMENT COMPANY XEROX Research Centre Europe



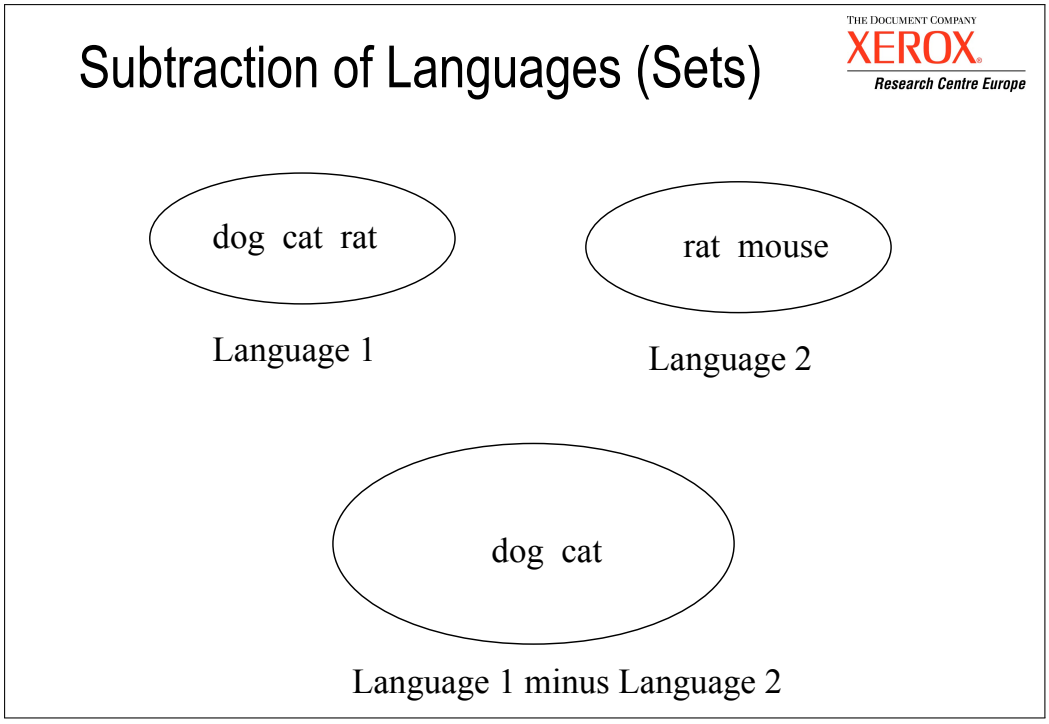
Quelle: B04



Quelle: B04

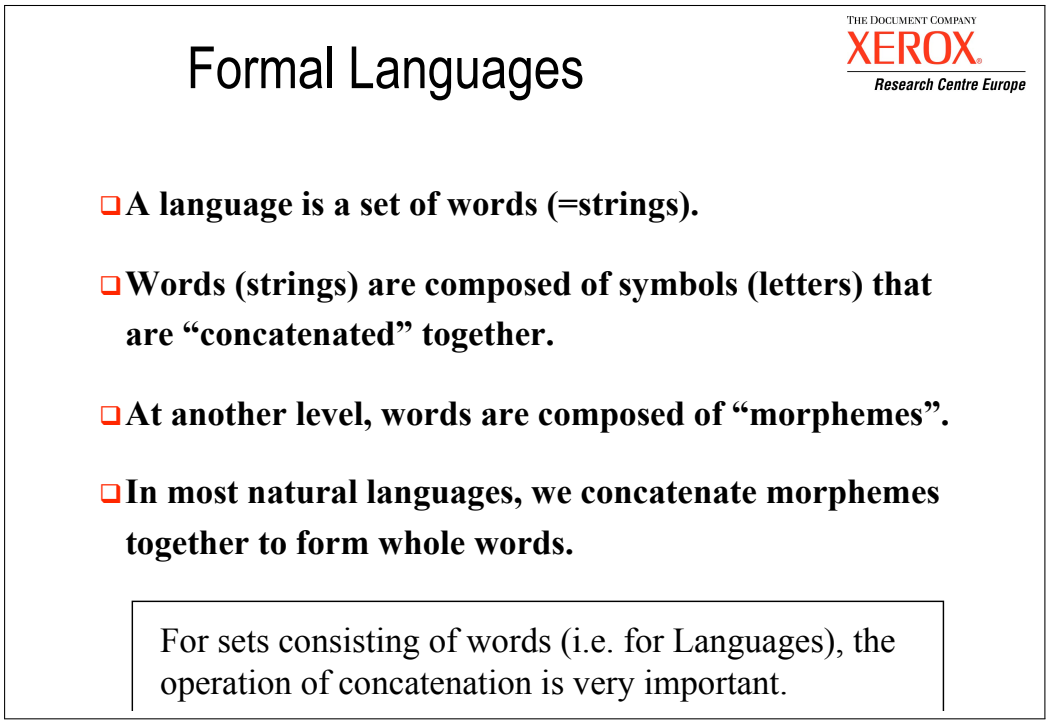


Quelle: B04



Quelle: B04

3.3.3 Konkatenation



Quelle: B04

Concatenation of Languages

THE DOCUMENT COMPANY XEROX Research Centre Europe

work talk walk

Root Language

0 ing ed s

Suffix Language

0 or ε denotes the empty string

work working
worked works talk
talking talked talks
walk walking
walked walks

The concatenation of the Suffix language after the Root language.

Quelle: B04

Concatenation of Languages II

THE DOCUMENT COMPANY XEROX Research Centre Europe

re out 0

Prefix Language

work talk walk

Root Language

0 ing ed s

Suffix Language

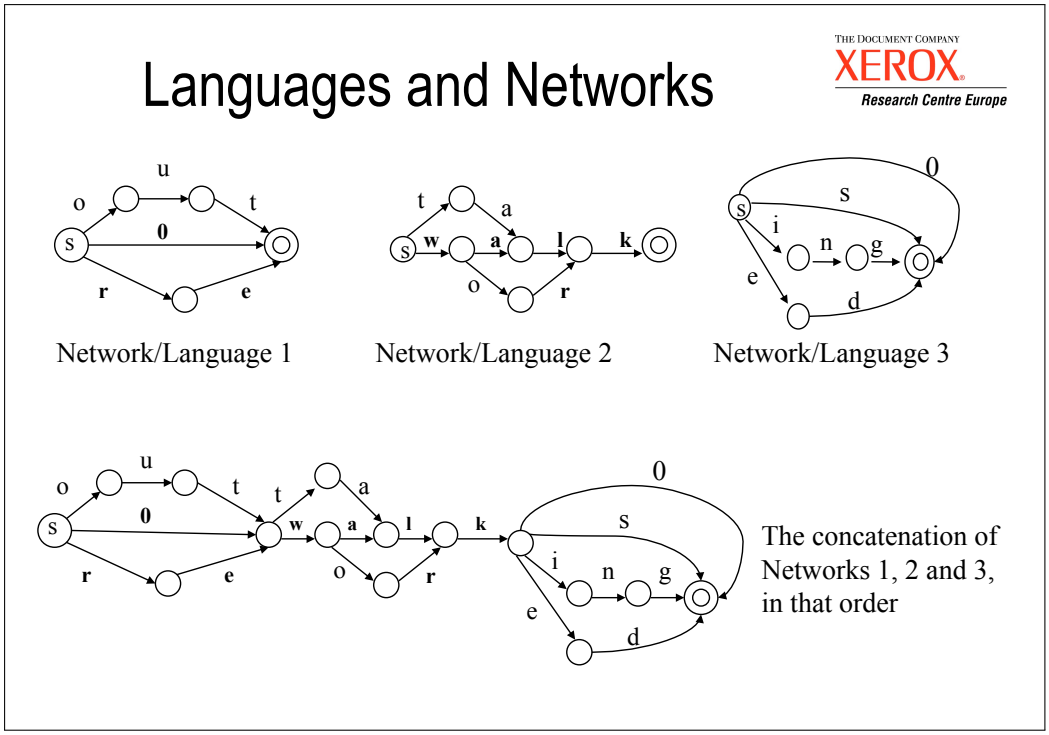
rework reworking
reworked reworks
retalk retalking
retalked retalks
rewalk rewalking
rewalked rewalks

outwork outworking
outworked outworks
outtalk outtalking
outtalked outtalks
outwalk outwalking
outwalked outwalks

work working
worked works talk
talking talked talks
walk walking
walked walks

The concatenation of the Prefix language, Root language, and the Suffix language.

Quelle: B04



Quelle: B04

3.4 Vertiefung

- Pflicht-Lektüre von Kapitel 1 „A Gentle Introduction“ aus Beesley/Karttunen 2003 (Materialordner) bis mindestens Seite 28
- Versuchen Sie mindestens Übung 1.10.1 „The Better Cola Machine“

Kapitel 4

Reguläre Ausdrücke für reguläre Sprachen

Lernziele

- Kenntnis über reguläre Sprachen, reguläre Ausdrücke und ihren Bezug zu endlichen Automaten
- Kenntnis der formalen Definition der 3 wichtigsten Typen von endlichen Automaten (DEA, NEA, ϵ-NEA)
- Kenntnis der grundlegenden regulären Ausdrücke und Operatoren in `xfst` und Wissen um den engen Bezug zur Theorie der endlichen Automaten
- Auffrischen der Kenntnisse über formalen Sprachen
- Fähigkeit, eine formale, rekursive Definition der Sprache eines Automaten wie $\delta*$ zu verstehen und formal sauber auswerten zu können

4.1 Endliche Automaten

4.1.1 Deterministische endliche Automaten

Deterministische Endliche Automaten (DEA)

Idee des akzeptierenden deterministischen endlichen Automaten
Ein endlicher Automat ist eine (abstrakte) Maschine zur zeichenweisen Erkennung von Wörtern einer regulären Sprache. Er ist nach jedem Verarbeitungsschritt in genau einem Zustand. Bei jedem Schritt wird ein Zeichen gelesen und aufgrund des aktuellen Zustands und dem Lesezeichen in einen Nachfolgezustand gewechselt. Wenn kein Zeichen mehr zu lesen ist und der Automat in einem Endzustand ist, gilt die gelesene Zeichenkette als akzeptiert. Wenn kein Übergang mit dem gelesenen Symbol möglich ist, gilt die zu verarbeitende Zeichenkette als nicht akzeptiert. Beim Einlesen des ersten Zeichens einer Zeichenkette ist der Automat immer im sogenannten Startzustand.

Definition 4.1.1 (DEA, *deterministic finite state automaton*)**.** Ein *deterministischer endlicher Automat* $A = \langle \Phi, \Sigma, \delta, S, F \rangle$ besteht aus

1. einer endlichen Menge *Zustände* Φ

2. einem endlichen *Eingabealphabet* Σ
3. einer (ev. partiellen) *Zustandsübergangsfunktion* $\delta : \Phi \times \Sigma \to \Phi$
4. einem *Startzustand* $S \in \Phi$
5. einer Menge von *Endzuständen* $F \subseteq \Phi$

Hinweis
Die Übergangsfunktion δ bestimmt *den* Folgezustand, der ausgehend vom aktuellen Zustand beim Lesen eines *einzelnen* Zeichens erreicht wird.

Zustandübergangsdiagramm
Es stellt einen EA anschaulich als gerichteten Graphen dar.

Beispiel 4.1.2.

$A = \langle \{0,1\}, \{a,b\}, \delta, 0, \{1\} \rangle$ mit $\delta = \{\langle\langle 0,a\rangle,1\rangle, \langle\langle 0,b\rangle,1\rangle, \langle\langle 1,a\rangle,1\rangle, \langle\langle 1,b\rangle,1\rangle\}$

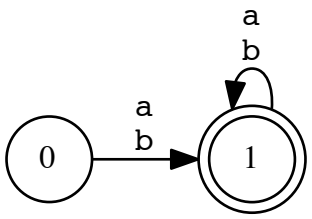
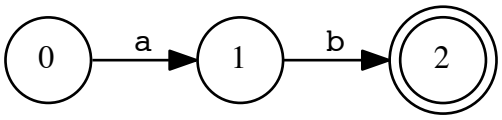


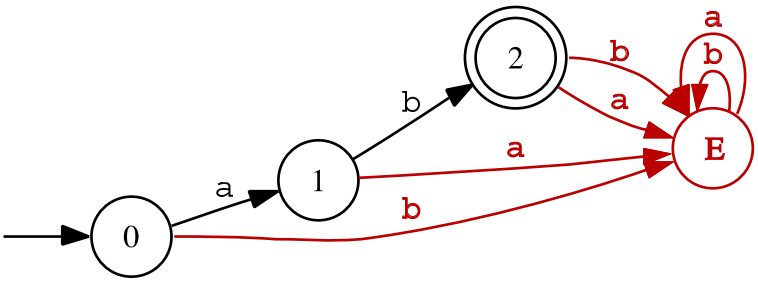
Abbildung 4.1: Zustandsübergangsdiagramm

- Jeder Zustand ist ein Knoten.
- Der Startzustand ist mit einem freien Pfeil oder auch nur mit der Beschriftung 0 (Null) bzw. S (Start) markiert.
- Endzustände sind doppelt umrahmt.
- Jeder Zustandsübergang ist eine gerichtete Kante vom aktuellen Zustand zum Folgezustand mit dem Zeichen als Kantenlabel.
- Alternative Zeichen werden oft mit einem Pfeil kondensiert dargestellt.

Partielle Zustandübergangsdiagramme
Die Definition von δ als totale Funktion verlangt für alle Zustände und Zeichen aus Σ eine Kante. In der Linguistik zeigt man oft nur Knoten und Kanten, die für das Akzeptieren einer Zeichenkette relevant sind.

Beispiel 4.1.3 (Partieller und vollständiger Automat).



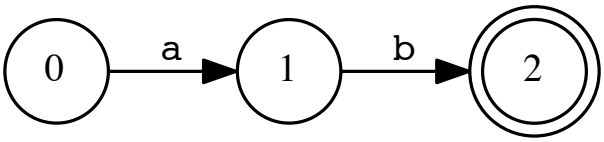


Fehlerzustand (*error state*) und Fehlerübergänge
Um ein partielles Übergangsdiagramm zu vervollständigen, füge einen Zustand $E \notin F$ dazu, und mache ihn zum Endknoten aller bisher nicht-bestehenden Kanten.

EA im XFST-Prolog-Format

Beispiel 4.1.4 (Direktes Einlesen von EA im PROLOG-Format).

```
xfst[0]: read prolog
        Type one or more networks in prolog format.
        Use ^D to terminate input.
arc(ex2,0,1,"a").
arc(ex2,1,2,"b").
final(ex2,2).
```



- Kantenrepräsentation: `arc(NET, VON, NACH, ZEICHEN)`
- `0` ist Startzustand.
- Endzustände: `final(NET, ZUSTAND)`
- Schreiben von EA Prolog-Format: `write prolog > Dateiname`
  Kurzform `wpl`
- Einlesen aus Datei: `read prolog < Dateiname`

Textuelle Ausgabe von EA

Beispiel 4.1.5 (Anzeige von Informationen zum obersten EA auf dem Stack).

```
xfst[1]: print net
Sigma: a b
Size: 2
Net: ex2
```

```
Flags: epsilon_free
Arity: 1
s0: a -> s1.
s1: b -> fs2.
fs2: (no arcs)
```

- *Sigma*: Auflistung aller Zeichen des Alphabets
- *Size*: Anzahl der Kantenbeschriftungen
- *Net*: Name des EA
- *Flags*: Wichtige Eigenschaften bzgl. ϵ, Sprachgrösse, EA-Typ usw.
- *Arity*: 1 = EA, 2 = Transduktor
- Zustände: `s0`: Startzustand; für $n > 0$ `fs`n: Endzustand, `s`n: Kein Endzustand
- `s1: b -> fs2.`: Vom Zustand `s1` geht eine mit `b` beschriftete Kante zum Folgezustand `fs2`.

4.1.2 Nicht-deterministische endliche Automaten

Nicht-deterministischer endlicher Automat (NEA)

Idee des akzeptierenden nicht-deterministischen endlichen Automaten
Im Gegensatz zum DEA kann ein NEA nach jedem Verarbeitungsschritt in mehr als einen aktuellen Zustand wechseln. Bei jedem Schritt wird ebenfalls ein Zeichen gelesen und aufgrund der möglichen aktuellen Zustände und dem Lesezeichen in die zulässigen Nachfolgezustände gewechselt.

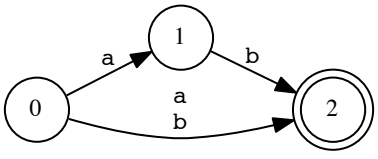
Beispiel-Evaluation
Siehe Abb. 4.1.2 auf Seite 36.

Definition 4.1.6 (NEA, *non-deterministic finite state automaton*)**.** Ein *nicht-deterministischer endlicher Automat* $A = \langle \Phi, \Sigma, \delta, S, F \rangle$ unterscheidet sich von einem DEA nur in der Übergangsfunktion. Vom aktuellen Zustand kann man mit einem Zeichen zu beliebig vielen Folgezuständen übergehen ($\wp$=Potenzmenge).

$$\delta : \Phi \times \Sigma \to \wp(\Phi)$$

Beispiel 4.1.7 (Tabellendarstellung und Zustandsdiagramm).

δ	a	b
s0	{s1,fs2}	{fs2}
s1	$\emptyset$	{fs2}
fs2	$\emptyset$	$\emptyset$



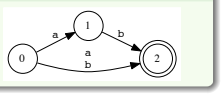
Beispiel-Berechnung für δ^*

1. $\delta^*(0, ab)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= \{T\} \\ \delta^*(T, wx) &= \bigcup_{T' \in \delta^*(T,w)} \delta(T', x)\end{aligned}$$

Beispiel-Automat



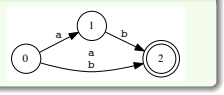
Beispiel-Berechnung für δ^*

1. $\delta^*(0, ab)$
2. $= \bigcup_{T' \in \delta^*(0,a)} \delta(T', b)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= \{T\} \\ \delta^*(T, wx) &= \bigcup_{T' \in \delta^*(T,w)} \delta(T', x)\end{aligned}$$

Beispiel-Automat



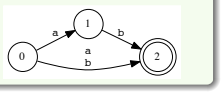
Beispiel-Berechnung für δ^*

1. $\delta^*(0, ab)$
2. $= \bigcup_{T' \in \delta^*(0,a)} \delta(T', b)$
3. $= \bigcup_{T' \in \bigcup_{T'' \in \delta^*(0,\epsilon)} \delta(T'', a)} \delta(T', b)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= \{T\} \\ \delta^*(T, wx) &= \bigcup_{T' \in \delta^*(T,w)} \delta(T', x)\end{aligned}$$

Beispiel-Automat



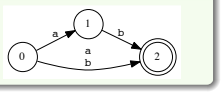
Beispiel-Berechnung für δ^*

1. $\delta^*(0, ab)$
2. $= \bigcup_{T' \in \delta^*(0,a)} \delta(T', b)$
3. $= \bigcup_{T' \in \bigcup_{T'' \in \delta^*(0,\epsilon)} \delta(T'', a)} \delta(T', b)$
4. $= \bigcup_{T' \in \bigcup_{T'' \in \{0\}} \delta(T'', a)} \delta(T', b)$
5. $= \bigcup_{T' \in \delta(0,a)} \delta(T', b)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= \{T\} \\ \delta^*(T, wx) &= \bigcup_{T' \in \delta^*(T,w)} \delta(T', x)\end{aligned}$$

Beispiel-Automat



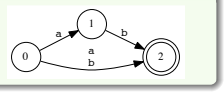
Beispiel-Berechnung für δ^*

1. $\delta^*(0, ab)$
2. $= \bigcup_{T' \in \delta^*(0,a)} \delta(T', b)$
3. $= \bigcup_{T' \in \bigcup_{T'' \in \delta^*(0,\epsilon)} \delta(T'', a)} \delta(T', b)$
4. $= \bigcup_{T' \in \bigcup_{T'' \in \{0\}} \delta(T'', a)} \delta(T', b)$
5. $= \bigcup_{T' \in \delta(0,a)} \delta(T', b)$
6. $= \bigcup_{T' \in \{1,2\}} \delta(T', b)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= \{T\} \\ \delta^*(T, wx) &= \bigcup_{T' \in \delta^*(T,w)} \delta(T', x)\end{aligned}$$

Beispiel-Automat



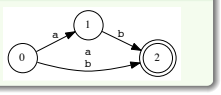
Beispiel-Berechnung für δ^*

1. $\delta^*(0, ab)$
2. $= \bigcup_{T' \in \delta^*(0,a)} \delta(T', b)$
3. $= \bigcup_{T' \in \bigcup_{T'' \in \delta^*(0,\epsilon)} \delta(T'', a)} \delta(T', b)$
4. $= \bigcup_{T' \in \bigcup_{T'' \in \{0\}} \delta(T'', a)} \delta(T', b)$
5. $= \bigcup_{T' \in \delta(0,a)} \delta(T', b)$
6. $= \bigcup_{T' \in \{1,2\}} \delta(T', b)$
7. $= \delta(1, b) \cup \delta(2, b)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= \{T\} \\ \delta^*(T, wx) &= \bigcup_{T' \in \delta^*(T,w)} \delta(T', x)\end{aligned}$$

Beispiel-Automat



Beispiel-Berechnung für δ^*

1. $\delta^*(0, ab)$
2. $= \bigcup_{T' \in \delta^*(0,a)} \delta(T', b)$
3. $= \bigcup_{T' \in \bigcup_{T'' \in \delta^*(0,\epsilon)} \delta(T'', a)} \delta(T', b)$
4. $= \bigcup_{T' \in \bigcup_{T'' \in \{0\}} \delta(T'', a)} \delta(T', b)$
5. $= \bigcup_{T' \in \delta(0,a)} \delta(T', b)$
6. $= \bigcup_{T' \in \{1,2\}} \delta(T', b)$
7. $= \delta(1, b) \cup \delta(2, b)$
8. $= \{2\} \cup \emptyset$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= \{T\} \\ \delta^*(T, wx) &= \bigcup_{T' \in \delta^*(T,w)} \delta(T', x)\end{aligned}$$

Beispiel-Automat

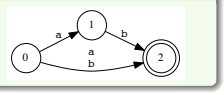


Abbildung 4.2: Beispielevaluation der erweiterten Übergangsfunktion für NEA

Idee des akzeptierenden nicht-deterministischen endlichen Automaten mit ϵ
Bei jedem Schritt wird wie beim NEA ein Zeichen gelesen und aufgrund der möglichen aktuellen Zustände und dem Lesezeichen in die zulässigen Nachfolgezustände gewechselt. Die aktuellen Zustände umfassen aber immer auch alle Zustände, welche durch beliebig viele ϵ-Übergange verbunden sind. Bei einem ϵ-Übergang wird kein Zeichen gelesen.

Definition 4.1.8 (ϵ-NEA, *non-deterministic finite state automaton*)**.** Ein *nicht-deterministischer endlicher Automat mit ϵ-Übergängen* $A = \langle \Phi, \Sigma, \delta, S, F \rangle$ unterscheidet sich von einem NEA nur in der Übergangsfunktion.

$$\delta : \Phi \times (\Sigma \cup \{\epsilon\}) \to \wp(\Phi)$$

QUIZ Leichtes QUIZ zu endlichen Automaten

4.2 Reguläre Ausdrücke in XFST

Reguläre Ausdrücke (RA) in XFST

- Das Ausdrucksmittel der regulären Ausdrücke (*regular expression calculus*) erlaubt eine konzise Beschreibung von regulären Sprachen.
- RA sind eine deklarative Programmiersprache für reguläre Sprache.
- RA bestehen aus Symbolen und Operatoren (einstellig bis mehrstellig), welche Symbole und RA verknüpfen.
- Die Bedeutung eines RA ergibt sich durch die Angabe der entsprechenden regulären Mengenoperation.
- `xfst` kompiliert einen RA in einen EA, der genau die Sprache des RA erkennt.

Beispiel 4.2.1 (Einlesen und Kompilieren eines RA mit xfst).

```
read regex RA ;                                    # EA kommt auf Stack
define varname RA ;                              # EA als Variable zugänglich
```

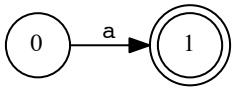
Beziehung zwischen RA, DEA und formalen Sprachen
Zu jedem regulären Ausdruck RA existiert mindestens ein EA, der die vom RA bezeichnete reguläre Sprache akzeptiert.

4.2.1 Zeichensymbole

Normales Zeichensymbol

Beispiel 4.2.2 (xfst und Zustandsdiagramm).

```
read regex a ;
```



Zulässige Einzelsymbole
Jedes der folgenden alphanumerischen Symbole x bezeichnet als regulärer Ausdruck (RA) die Sprache $\{x\}$:

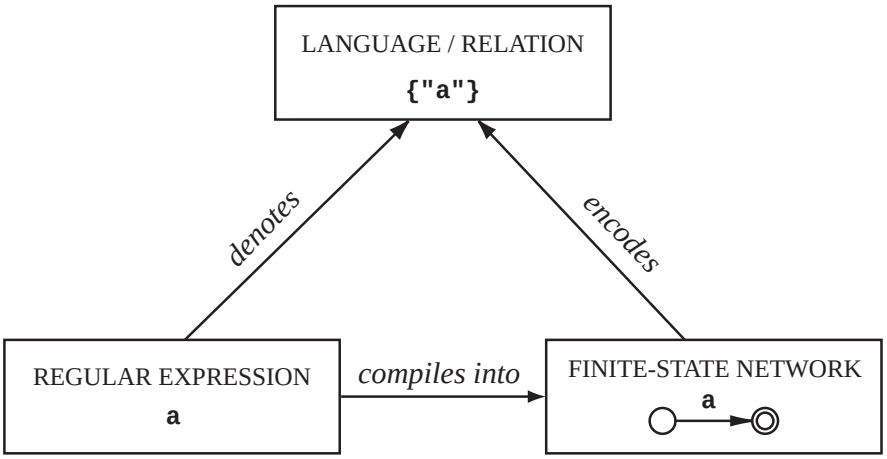
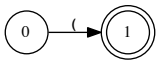


Abbildung 4.3: Beziehung zwischen formalen Sprachen, regulären Ausdrücken und endlichen Automaten (aus [Beesley und Karttunen 2003b])

- Alle Klein- und Grossbuchstaben (inklusive diakritische Zeichen mit Kodes > 128)
- Alle Ziffern ausser der Null.

%-geschütztes Zeichensymbol (*escape symbol*)

Beispiel 4.2.3 (xfst und Zustandsdiagramm).
`read regex %( ;`



Zeichen mit Sonderbedeutung
Das Zeichen `%` gefolgt von einem der folgenden Symbole x

`0 ? + * | . , : ; - / % ~ ( ) [ ] { } " \`

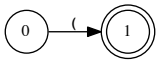
bezeichnet als RA die Sprache $\{x\}$.

Hinweise

- Wenn Zeichen ohne Sonderbedeutung geschützt werden, stehen sie weiterhin für sich selbst.
- Nach % wörtlich eingetipptes Leerzeichen, Tabulator oder Zeilenwechsel sind ebenfalls geschützt.

Zitiertes Zeichensymbol

Beispiel 4.2.4 (xfst und Zustandsdiagramm).
`read regex "(" ;`



Mehrzeichensymbol
Jedes Zeichen x zwischen doppelten Hochkommata ausser

\, " und getippter Zeilenwechsel

bezeichnet als RA die Sprache $\{x\}$.

Hinweis
Zwischen Hochkommata verlieren alle Zeichen ihre Sonderbedeutung.

Backslash-Notation zwischen doppelten Hochkommata
Backslash-Notation für *nicht-druckbare Zeichen* und *numerische Zeichenkodes* sind zwischen doppelten Hochkommata möglich in xfst.

- `"\t"` (Tabulator), `"\n"`, `"\r"` (Zeilenwechsel), `"\""` (doppelte Hochkommata), `"\\"` (Backslash)
- 8bit-Zeichenkode in Hexadezimalnotation: `"\x`*HH*`"`
- 16bit-Unicode in Hexadezimalnotation: `"\u`*HHHH*`"`

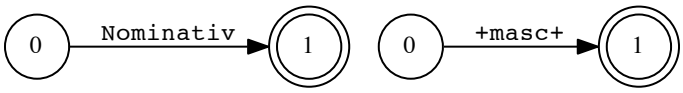
Hinweise

- `foma` kennt nur `\u`-Notation! Aus Kompatibilität nichts anderes verwenden!
- Für *H* in Hexadezimalnotation können die Zeichen 0-9, A-F, a-f verwendet werden.
- Um in `xfst` ISO-8859-1 zu verwenden, kann der Standardzeichensatz umgestellt werden (oder Kodierungskommentare verwenden) `xfst: set char-encoding ISO-8859-1`

Symbole aus mehreren Zeichen (*multicharacter symbols*)

Beispiel 4.2.5 (xfst und Zustandsdiagramm).

```
read regex Nominativ ;
read regex %+masc%+ ;
```

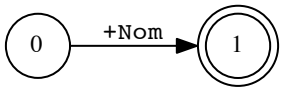


Mehrzeichensymbol
Jede Folge von normalen Zeichen oder %-geschützten Zeichen ist ein Mehrzeichensymbol x, das als RA die Sprache $\{x\}$ bezeichnet.

Wichtig
Mehrzeichensymbole werden von einem endlichen Automaten in einem einzigen Zustandübergang konsumiert.

Zitierte Mehrzeichensymbole
Auch Mehrzeichensymbole lassen sich mit doppelten Hochkommata zitieren. Es gelten die gleichen Regeln wie bei den Einzelzeichensymbole. In zitierten Symbolen verliert % seine Rolle als Escape-Zeichen.

Beispiel 4.2.6 (xfst und Zustandsdiagramm).
```
read regex "+Nom";
```



Guter Programmierstil

- zitiert alle Mehrzeichensymbole.
- verwendet in allen Mehrzeichensymbolen Sonderzeichen, welche im normalen Anwendungsgebiet nicht vorkommen.
- verwendet diese Symbole für morphosyntaktische Merkmale und Grenzen auf der Analyseebene

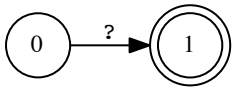
Epsilon-Symbol
Das Ziffern-Zeichen `0` ist ein Spezial-Symbol, das als RA die Sprache $\{\epsilon\}$ bezeichnet.

Beispiel 4.2.7 (xfst und Zustandsdiagramm).
```
read regex 0 ;
```

Hinweise
Um die Ziffer `0` als Einzelzeichen zu erhalten, muss sie geschützt oder zitiert werden.
Die Sprache $\{\epsilon\}$ kann auch mit `""` oder `[]` notiert werden.

***Any*-Symbol**
Das Zeichen `?` ist ein Spezial-Symbol, das als RA die Sprache aller Zeichenketten aus 1 Symbol bezeichnet.

Beispiel 4.2.8 (xfst und Zustandsdiagramm).
```
read regex ? ;
```



Hinweise

- Der reguläre Ausdruck `[ ?* ]` bezeichnet die *universale Sprache* und diese entspricht Σ^*.
- Mehrzeichensymbole erweitern den Vorrat an Symbolen (das Alphabet) unbeschränkt über den Zeichensatz hinaus.

4.2.2 Operatoren

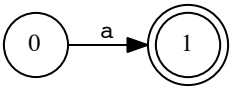
Reguläre Operatorenausdrücke in `xfst`
Wenn die beiden RA `A` und `B` die Sprachen A bzw. B bezeichnen, dann bezeichnet der RA

- `[ A B ]` die Sprache $A \bullet B$ (Verkettung)
- `[ A * ]` die Sprache A^* (beliebige Verkettung)
- `[ A + ]` die Sprache A^+ (Wiederholung)
- `[ A ^ n ]` die Sprache A^n (n-fache Verkettung)
- `[ A | B ]` die Sprache $A \cup B$ (Vereinigung)
- `[ A & B ]` die Sprache $A \cap B$ (Schnitt)
- `[ A - B ]` die Sprache $A \setminus B$ (Mengendifferenz)
- `( A )` die Sprache $A \cup \{\epsilon\}$ (Optionalität)

Hinweise zur Operator-Syntax

Beispiel 4.2.9 (Präzedenz von Operatoren). Welcher der beiden RA entspricht dem Zustandsdiagramm?

```
read regex a | b & c ;
read regex a | [ b & c ] ;
```



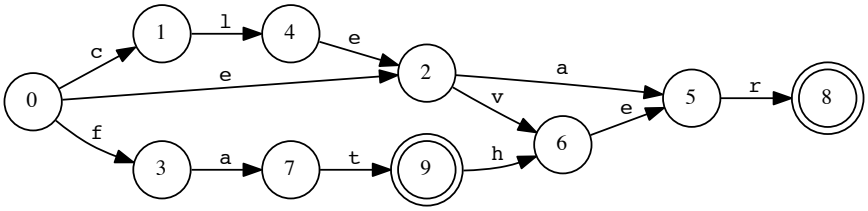
Hinweise

- Die eckigen Klammern können weggelassen werden, wenn die Priorität der Operatoren beachtet wird.
- In abnehmender Bindungsstärke geordnet: `+ * ^ ␣ | &` (␣ steht für Konkatenationsoperator)
- Weder runde noch geschweifte (`{}`) Klammern dürfen zur Gruppierung verwendet werden.

Klammernotation für Konkatenation
Jeder RA wie `{TEXT}` steht für die Konkatenation aller Einzelzeichen, die zwischen den geschweiften Klammern notiert sind: `[T E X T]`.

Beispiel 4.2.10 (xfst und Zustandsdiagramm).

```
read regex {clear}|{clever}
           |{ear}|{ever}
           |{fat}|{father};
```



Hinweis
Endliche Automaten speichern Lexika kompakter als Suchbäume (Buchstabenbäume, Trie). Grund: Nicht bloss gemeinsame gemeinsamen Präfixe, sondern auch gemeinsame Suffixe sind nur einmal repräsentiert.

Operatoren für Komplemente
Wenn der RA `A` die Sprache A über Σ bezeichnet, dann bezeichnet der RA

- `[ ~ A ]` die Sprache $\Sigma^* \setminus A$ (Sprach-Komplement, alle *Zeichenketten* ausser jene in A)
- `[ \ A ]` die Sprache $\Sigma \setminus A$ (Alphabet-Komplement, alle *Symbole* ausser jene in A)

Universale Sprache, Komplement und Subtraktion
Für das Sprach-Komplement von A gilt die folgende Äquivalenz

`[ ~ A ]` = `[?* - A ]`

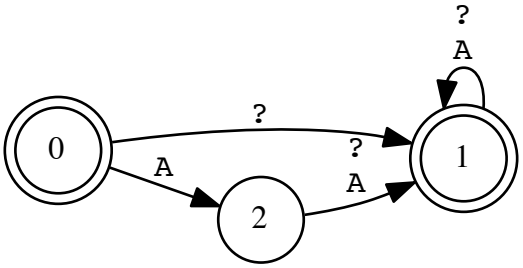
Was gilt für das Komplement des Alphabets?

`[ \ A ]` =

`?` als RA-Symbol vs. `?` als Kantenbeschriftung in EA
Das ANY-Symbol `?` steht als RA für die Sprache aller Zeichenketten der Länge 1.

Beispiel 4.2.11 (Interpretation von `~ A`).

```
xfst[0]: read regex [~ A] ;
xfst[1]: print net
Sigma: ? A
Size: 2.
Flags: deterministic, pruned, minimized, epsilon_free
Arity: 1
fs0: ? -> fs1, A -> s2.
fs1: ? -> fs1, A -> fs1.
s2: ? -> fs1, A -> fs1.
```



Im regulären Ausdruck `?*`, der die universale Sprache bezeichnet, bedeutet das Fragezeichen nicht dasselbe wie als Kantenbeschriftung. Die Verwendung des gleichen Zeichens für unterschiedliche Zwecke mag verwirren. Die Funktion als Kantenbeschriftung und als regulärer Ausdruck lässt sich aber immer leicht unterscheiden.

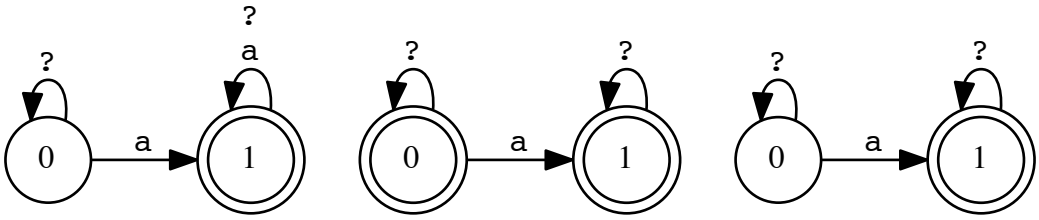
Definition 4.2.12 (UNKNOWN-Symbol)**.** Das *UNKNOWN-Symbol* `?` steht als Kantenbeschriftung für diejenigen Symbole des Eingabealphabets, welche nicht explizit im Sigma aufgeführt sind (dem Sigma unbekannt). Bei foma wird für das UNKNOWN-Symbol `@` verwendet.

Wozu kann es nützlich sein, das Alphabet nicht vollständig aufzählen zu müssen?

Operatoren für Enthalten-Sein (*contains*)
Wenn der RA `A` die Sprache A über Σ bezeichnet, dann bezeichnet der RA

- `[ $ A ]` die Sprache $\Sigma^* \bullet A \bullet \Sigma^*$ (Sprache, welche alle Zeichenketten von A als Teilzeichenkette enthält)
- `[ $? A ]` die Sprache, welche Teilzeichenketten von A höchstens einmal enthält.
- `[ $. A ]` die Sprache, welche Teilzeichenketten von A genau einmal enthält.

Beispiel 4.2.13 (Welches Zustandsdiagramm für welchen Operator?).



Operator für Beschränkung via Kontext
Der RA `[ A => L _ R ]` (*expression restriction*) restringiert alle Vorkommen von Teilzeichenketten a aus `A`. Jedem a muss eine Zeichenkette aus `L` vorangehen und eine Zeichenkette aus `R` nachfolgen.

Beispiel 4.2.14 (Ausdrucksrestriktion). Die Sprache von `[ a => b _ c ]` umfasst das Wort `back-to-back` oder `bc`, aber nicht das Wort `cab` oder `pack`.

Schrittweise Definition von `[ A => L _ R ]`

```
 define Maolv [~[?* L]] A ?* ; # Mindestens ein A ohne L vorher
define Maorn ?* A [~[R ?*]] ; # Mindestens ein A ohne R nachher
define WederMaolvNochMaorn ~Maolv & ~Maorn ;
```

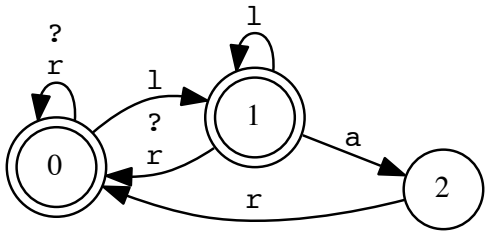
Hinweis
Die Sprache des Restriktions-Ausdrucks schränkt nur diejenigen Zeichenketten ein, welche als Teilzeichenkette ein Wort a aus A enthalten.

Wortende verankern in der Kontext-Beschränkung

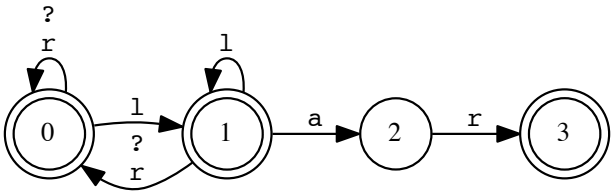
Die Spezialmarkierung für Wortende
Die Kontexte in `[ A => L _ R ]` sind gegen Aussen implizit mit der universalen Sprache konkateniert: `[ A => ?* L _ R ?*]`.

Die Spezialmarkierung `.#.` bedeutet in den Kontexten Verankerung am Wortende und verhindert so die implizite Verkettung. Im EA ist `.#.` nicht vorhanden, es kann aber wie ein Symbol in die RA eingefügt werden.

Beispiel 4.2.15 (Die Wirkung von `.#.`).



`[ a => l _ r ]`



`[ a => l _ r .#. ]`

Operator für Beschränkung des Kontexts
Der RA `[ A <= L _ R ]` (*context restriction*) erlaubt Kontexte aus L gefolgt von R nur, wenn dazwischen eine Teilzeichenketten a aus `A` vorkommt.

Beispiel 4.2.16 (Kontextrestriktion). Die Sprache von `[ a <= b _ c ]` umfasst das Wort `back-to-back` oder `cb`, aber nicht das Wort `bc` oder `bxc`.

Hinweis zu Restriktion
Der Operator `[ A <=> L _ R ]` vereinigt die Beschränkungen. A ist ausserhalb von L und R verboten (*expression restriction*) und der Kontext `L _ R` ist verboten, ausser ein String aus A steht dazwischen (*context restriction*).

4.3 Formale Sprachen

4.3.1 Zeichen und Zeichenketten

Das Alphabet (Sigma), Zeichen und Zeichenketten

Definition 4.3.1. Ein *Alphabet* ist eine endliche Menge von Zeichen (atomare Symbole). Es wird mit Σ (Sigma) notiert.

Definition 4.3.2. Eine *Zeichenkette* (Wort, *string*) von n Zeichen aus Σ ist eine endliche Folge der Länge n über Σ.

Die *leere Zeichenkette* (leeres Wort) ist die Folge von 0 Zeichen. Sie wird mit ϵ (Epsilon) notiert und hat die Länge 0.

Hinweis zur Notation
Eine Zeichenkette wird typischerweise durch Nebeneinanderschreiben (Juxtaposition) der Zeichen von links nach rechts notiert.
Sei $\Sigma = \{a, b\}$, dann sind etwa ϵ, a, bb oder $ababbba$ Wörter über Σ.

Stern von Sigma und formale Sprachen

Definition 4.3.3. Der *Stern von Sigma* ist die Menge aller Wörter über einem Alphabet Σ. Der Stern wird als Postfix-Operator Σ^* (sprich «Sigma Stern») notiert.

Sigma Stern erzeugt aus einer Symbolmenge eine Menge von Zeichenketten über diesen Symbolen.

Definition 4.3.4. Eine *formale Sprache* L über Σ ist eine Teilmenge des Sterns von Sigma.

$$L \subseteq \Sigma^*$$

Beispiel 4.3.5. <+>Sei $\Sigma = \{a\}$, dann ist $\Sigma^* = \{\epsilon, a, aa, aaa, \ldots\}$. Die Mengen $L_1 = \{\epsilon, a\}$ oder $L_2 = \{aa, aaaa, aaaaaa\}$ sind formale Sprachen, da sie (echte) Teilmengen von Σ^* sind.

Leere Sprachen vs. leere Zeichenkette

Hinweise

- Die *leere Sprache* ist die leere Menge, notiert als $\{\}$ oder $\emptyset$.
- Die Sprache, welche nur die *leere Zeichenkette* umfasst, wird als $\{\epsilon\}$ notiert.
- Die leere Sprache $\{\}$ und die Sprache $\{\epsilon\}$ sind nicht dasselbe.

Fragen

1. Ist $\{\}$ eine Sprache über jedem Σ?
2. Ist die Sprache $\{\epsilon\}$ Teilmenge jeder nicht-leeren Sprache?
3. Ist Σ^* eine Sprache über Σ?

4.3.2 Konkatenation

Konkatenation von Zeichenketten und Sprachen

Definition 4.3.6. Die *Konkatenation von Zeichenketten* ist eine zweistellige Funktion, welche ihre Argumente zu einem Wort verkettet. Für alle $u, v \in \Sigma^*$:

$$\bullet : \Sigma^* \times \Sigma^* \to \Sigma^*, \quad u \bullet v = uv$$

Definition 4.3.7. Die *Konkatenation von Sprachen* erweitert Verkettung auf Mengen von Zeichenketten. Für alle $M, N \subseteq \Sigma^*$:

$$\bullet : \wp(\Sigma^*) \times \wp(\Sigma^*) \to \wp(\Sigma^*), \quad M \bullet N = \{u \bullet v \mid u \in M \wedge v \in N\}$$

Eigenschaften der Konkatenation

Zeichenketten
Die Konkatenation ist *assoziativ* und hat ϵ als *neutrales Element*. Für alle $u, v, w \in \Sigma^*$:

$$u \bullet (v \bullet w) = (u \bullet v) \bullet w, \quad \epsilon \bullet u = u, \quad u \bullet \epsilon = u$$

Sprachen
Die Konkatenation ist *assoziativ* und hat $\{\epsilon\}$ als *neutrales Element*. Für alle $M, N, P \subset \Sigma^*$:

$$M \bullet (N \bullet P) = (M \bullet N) \bullet P, \quad \{\epsilon\} \bullet M = M, \quad M \bullet \{\epsilon\} = M$$

Klammerung
Die Assoziativität der Konkatenation macht Klammern optional.

Stern

Stern einer Sprache (*Kleene star*)

Definition 4.3.8 (N-fache Wiederholung)**.** <+>Die n-fache *Konkatenation einer Sprache L mit sich selbst* in der Potenznotation sei rekursiv definiert. Für $n \geq 1, n \in \mathbb{N}$:

$$\begin{aligned} L^0 &= \{\epsilon\} \\ L^n &= L \bullet L^{n-1} \end{aligned}$$

Definition 4.3.9. Der *Stern einer Sprache* (Kleene-Hülle) ist Vereinigung ihrer Konkatenationen.

$$^* : \wp(\Sigma^*) \to \wp(\Sigma^*), \quad L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots = \bigcup_{n \geq 0} L^n$$

Plus einer Sprache

Beispiel 4.3.10. <+>Sei $M = \{ab, c\}$ über $\Sigma = \{a, b, c\}$. Welche Zeichenketten enthält M^*?

Hinweis
Formale Sprachen sind Mengen und somit können alle Mengenoperationen wie Schnitt ($M \cap N$), Vereinigung ($M \cup N$), Mengendifferenz ($M \setminus N$) mit ihnen verwendet werden.

Definition 4.3.11. Das *Plus einer Sprache* (positive Kleene Hülle) ist

$$^+ : \wp(\Sigma^*) \to \wp(\Sigma^*), \quad L^+ = L^* \setminus \{\epsilon\}$$

4.3.3 Reguläre Sprachen

Reguläre Sprachen

Definition 4.3.12. Eine *Sprache* über $\Sigma = \{a_1, a_2, ..., a_n\}$ heisst *regulär*, wenn sie durch folgende reguläre Mengenausdrücke beschrieben werden kann:

- Die leere Menge $\emptyset$ ist regulär.
- Die Menge $\{\epsilon\}$ ist regulär.

- Die Mengen $\{a_i\}$ $(1 \leq i \leq n)$ sind regulär.
- Wenn L_1 und L_2 regulär sind, dann auch $(L_1 \cup L_2)$.
- Wenn L_1 und L_2 regulär sind, dann auch $(L_1 \bullet L_2)$.
- Ist L regulär, dann auch L^*.

Diese 6 Bildungsregeln reichen aus, um alle regulären Sprachen zu beschreiben. Meistens werden für die Kürze der Notation noch weitere Konstrukte definiert, welche sich aber immer auf obige 6 Bildungsregeln zurückführen lassen müssen.

Abschlusseigenschaften regulärer Sprachen

Definition 4.3.13. Eine Menge M heisst *abgeschlossen* (*closed under*) bezüglich einer Operation oder Funktion, wenn das Ergebnis der Operation mit Elementen aus M immer ein Element aus M ist.

Abschlusseigenschaften regulärer Sprachen

- Über der Menge der regulären Sprachen sind die Operationen $L_1 \bullet L_2$, L^* und $L \cup L_2$ per Definition abgeschlossen.
- Der Abschluss bezüglich der Operationen $L_1 \cap L_2$, $L_1 \setminus L_2$, L^+ und Komplementmenge $\overline{L}$ lässt sich beweisen.

QUIZ Leichtes QUIZ zu Zeichenketten, Sprachen und Konkatenation

4.4 Sprachen von EA

Reguläre Sprachen und EA
Die Menge der regulären Sprachen ist gleich der Menge der von EA akzeptierten Sprachen.

Wichtige in `xfst` berechenbare Eigenschaften von EA
Tests bezüglich endlicher Automaten auf dem Stack

- Ist eine Zeichenkette w *Element* der Sprache eines EA?
  `xfst[1]: apply up w` $w \in L(EA)$
- Ist die Sprache von EA die *leere Sprache*?
  `xfst[1]: test null` $L(EA) = \emptyset$
- Ist die Sprache von EA die *universale Sprache*?
  `xfst[1]: test lower-universal` $L(EA) = \Sigma^*$
- *Weitere Tests* zeigt `help text` in `xfst`/`foma`.

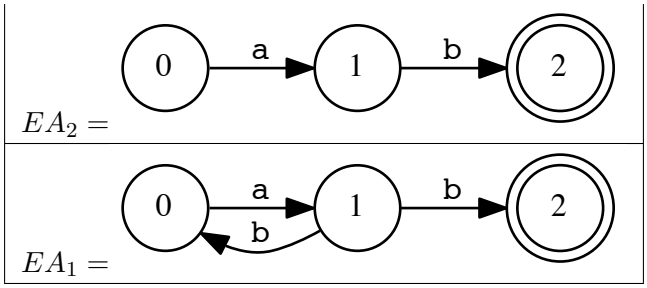


Abbildung 4.4: EA_2 über EA_1 auf dem Stapel

Wichtige in `xfst` berechenbare Beziehungen von EA
Tests bezüglich endlicher Automaten auf dem Stack (EA_2 auf EA_1)

- Besitzen EA_1 und EA_2 die *gleiche Sprache*?
  `xfst[2]: test equivalent` $L(EA_2) = L(EA_1)$
- Ist die Sprache von EA_2 *Teilmenge* der Sprache des darunter liegenden EA_1?
  `xfst[2]: test sublanguage` $L(EA_2) \subseteq L(EA_1)$

Anstelle von `apply up` kann man auch `apply down` sagen. Statt `test lower-universal` auch `upper-universal`. Diese Sprechweisen hängen mit den Transduktoren zusammen, welche eine "obere" und "untere" Sprache haben. Bei den endlichen Automaten sind aber beide Sprachen gleich.

4.4.1 DEA

Sprache eines DEA

Definition 4.4.1 (Zustandsübergangsfunktion für Zeichenketten)**.** Die *auf Zeichenketten erweiterte Übergangsfunktion* $\delta^* : \Phi \times \Sigma^* \to \Phi$ bestimmt den Zustand nach dem vollständigen Lesen eines Wortes.

Für alle $T \in \Phi, w \in \Sigma^*, x \in \Sigma$ eines DEA $A = \langle \Phi, \Sigma, \delta, S, F \rangle$

$$\begin{aligned} \delta^*(T, \epsilon) &= T \\ \delta^*(T, wx) &= \delta(\delta^*(T, w), x) \end{aligned}$$

Definition 4.4.2. Die *Sprache* $L(A)$ eines DEA $A = \langle \Phi, \Sigma, \delta, S, F \rangle$ ist die Menge aller Zeichenketten, für die δ^* ausgehend vom Startzustand einen Endzustand ergibt.

$$L_A = \{w \in \Sigma^* \mid \delta^*(S, w) \in F\}$$

Beispiel: Rekursive Berechnung von δ^* für DEA aus Beispiel 4.1.2

1. $\delta^*(0, aba)$
2. $= \delta(\delta^*(0, ab), a)$
3. $= \delta(\delta(\delta^*(0, a), b), a)$
4. $= \delta(\delta(\delta(\delta^*(0, \epsilon), a), b), a)$
5. $= \delta(\delta(\delta(0, a), b), a)$

6. $= \delta(\delta(1, b), a)$
7. $= \delta(1, a)$
8. $= 1$

Beispiel-Evaluation
Siehe Abb. 4.5 auf Seite 52.

4.4.2 NEA

Sprache von NEA

Definition 4.4.3 (Zustandsübergangsfunktion für Zeichenketten)**.** Die *auf Zeichenketten erweiterte Übergangsfunktion* $\delta^* : \Phi \times \Sigma^* \to \wp(\Phi)$ bestimmt die Menge der erreichbaren Zustände nach dem vollständigen Lesen einer Zeichenkette.

Für alle $T \in \Phi, w \in \Sigma^*, x \in \Sigma$ eines NEA $A = \langle \Phi, \Sigma, \delta, S, F \rangle$

$$\begin{aligned} \delta^*(T, \epsilon) &= \{T\} \\ \delta^*(T, wx) &= \bigcup_{T' \in \delta^*(T,w)} \delta(T', x) \end{aligned}$$

Definition 4.4.4. Die *Sprache* $L(A)$ eines NEA $A = \langle \Phi, \Sigma, \delta, S, F \rangle$ ist die Menge aller Zeichenketten, für die δ^* ausgehend vom Startzustand mindestens einen Endzustand enthält.

$$L_A = \{w \in \Sigma^* \mid (\delta^*(S, w) \cap F) \neq \emptyset\}$$

Beispiel-Berechnung für δ^*

1. $\delta^*(0, ab)$
2. $= \bigcup_{T' \in \delta^*(0,a)} \delta(T', b)$
3. $= \bigcup_{T' \in \bigcup_{T'' \in \delta^*(0,\epsilon)} \delta(T'', a)} \delta(T', b)$
4. $= \bigcup_{T' \in \bigcup_{T'' \in \{0\}} \delta(T'', a)} \delta(T', b)$
5. $= \bigcup_{T' \in \delta(0,a)} \delta(T', b)$
6. $= \bigcup_{T' \in \{1,2\}} \delta(T', b)$
7. $= \delta(1, b) \cup \delta(2, b)$
8. $= \{2\} \cup \emptyset$
9. $= \{2\}$

Sprache von NEA mit ϵ

Definition 4.4.5 (Zustandsübergangsfunktion für Zeichenketten)**.** Die *auf Zeichenketten erweiterte Übergangsfunktion* $\delta^* : \Phi \times \Sigma^* \to \wp(\Phi)$ bestimmt die Menge der erreichbaren Zustände nach dem vollständigen Lesen einer Zeichenkette.

Für alle $T \in \Phi, w \in \Sigma^*, x \in \Sigma$ eines ϵ-NEA $A = \langle \Phi, \Sigma, \delta, S, F \rangle$

$$\begin{aligned} \delta^*(T, \epsilon) &= \delta_\epsilon^*(T) \\ \delta^*(T, wx) &= \bigcup_{T' \in \delta(\delta^*(T,w),x)} \delta_\epsilon^*(T') \end{aligned}$$

Die Bestimmung der Übergangsfunktion δ^* auf Zeichenketten setzt auf der reflexiven und transitiven Hülle der ϵ-Übergänge auf: δ_ϵ^*.

Reflexive und transitive Hülle der ϵ-Übergänge

Definition 4.4.6. Die *ϵ-Hülle* $\delta_\epsilon^* : \Phi \to \wp(\Phi)$ eines ϵ-NEA für einen Zustand $T \in \Phi$ ist rekursiv definiert durch:

- *Reflexive* Hüllenbildung: $T \in \delta_\epsilon^*(T)$
- *Transitive* Hüllenbildung (rekursiv): Falls $T_1 \in \delta_\epsilon^*(T)$ und $T_2 \in \delta(T_1, \epsilon)$, dann $T_2 \in \delta_\epsilon^*(T)$.

4.4.3 Äquivalenz von EAs

Äquivalenz der Sprachen von DEA, NEA und ϵ-NEA

Entfernung von ϵ-Übergängen
Für jeden ϵ-NEA lässt sich ein NEA (ohne ϵ) konstruieren, der die gleiche Sprache akzeptiert.
`xfst[1]: epsilon-remove net`

Entfernung von Nicht-Determinismus
Für jeden NEA lässt sich ein DEA konstruieren, der die gleiche Sprache akzeptiert.
`xfst[1]: determinize net`

Entfernung unnötiger Zustände
Für jeden DEA lässt sich ein DEA mit minimaler Anzahl Zustände konstruieren, der die gleiche Sprache akzeptiert.
`xfst[1]: minimize net`

Der Befehl `minimize net` beinhaltet die Entfernung von ϵ-Übergängen und die Determinisierung. D.h. ein minimisierter EA ist immer deterministisch und ohne ϵ-Übergängen.

Beispiel für äquivalente EA
Siehe Abb. 117 auf Seite 53.

4.5 Vertiefung

- Ein sehr interaktive Einführung in die Automatentheorie findet sich unter `http://www.jflap.org`.
- Jede Einführung in die theoretische Informatik behandelt endliche Automaten und ihre Sprachen. Meine Formalisierungen folgen [Carstensen et al. 2004, Kapitel 2.2], welche einfache linguistisch motivierte Beispiele beinhaltet.
- Die Formalisierung der Sprache der ϵ-NEA in [Kaplan und Kay 1994] und [Roche und Schabes 1996] (die Version mit δ^*) ist m. E. induktiv nicht besonders gut gelöst, deshalb verwende ich dort den Klassiker [Hopcroft et al. 2002]. Dieses Buch bietet u.a. eine sorgfältig begründende Einführung in die Theorie und die wichtigen Algorithmen im Umfeld von EA.
- Dass die Entstehung des Konzepts der endlichen Automaten erstaunlicherweise viel mit dem Modell der Nervenzellen von McCulloch und Pitts zu tun hat, kann in [Lakovic und Mai 1999] nachgelesen werden.
- Hinweise zur Definition der Operatoren: Im Aufsatz [Cohen-Sygal und Wintner 2005] werden xfst-Konstrukte in den Formalismus der freiverfügbaren Prolog-basierten FSA-Tools von G. van Noord übersetzt.

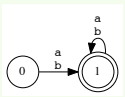
Beispiel: Rekursive Berechnung von δ^* für DEA

1. $\delta^*(0, aba)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= T \\ \delta^*(T, wx) &= \delta(\delta^*(T, w), x)\end{aligned}$$

Beispiel-Automat



EA RA Sprachen Sprachen von EA DEA NEA ϵ-NEA Äquivalenz

Beispiel: Rekursive Berechnung von δ^* für DEA

1. $\delta^*(0, aba)$
2. $= \delta(\delta^*(0, ab), a)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= T \\ \delta^*(T, wx) &= \delta(\delta^*(T, w), x)\end{aligned}$$

Beispiel-Automat

EA RA Sprachen Sprachen von EA DEA NEA ϵ-NEA Äquivalenz

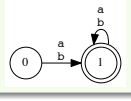
Beispiel: Rekursive Berechnung von δ^* für DEA

1. $\delta^*(0, aba)$
2. $= \delta(\delta^*(0, ab), a)$
3. $= \delta(\delta(\delta^*(0, a), b), a)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= T \\ \delta^*(T, wx) &= \delta(\delta^*(T, w), x)\end{aligned}$$

Beispiel-Automat



EA RA Sprachen Sprachen von EA DEA NEA ϵ-NEA Äquivalenz

Beispiel: Rekursive Berechnung von δ^* für DEA

1. $\delta^*(0, aba)$
2. $= \delta(\delta^*(0, ab), a)$
3. $= \delta(\delta(\delta^*(0, a), b), a)$
4. $= \delta(\delta(\delta(\delta^*(0, \epsilon), a), b), a)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= T \\ \delta^*(T, wx) &= \delta(\delta^*(T, w), x)\end{aligned}$$

Beispiel-Automat

EA RA Sprachen Sprachen von EA DEA NEA ϵ-NEA Äquivalenz

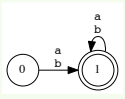
Beispiel: Rekursive Berechnung von δ^* für DEA

1. $\delta^*(0, aba)$
2. $= \delta(\delta^*(0, ab), a)$
3. $= \delta(\delta(\delta^*(0, a), b), a)$
4. $= \delta(\delta(\delta(\delta^*(0, \epsilon), a), b), a)$
5. $= \delta(\delta(\delta(0, a), b), a)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= T \\ \delta^*(T, wx) &= \delta(\delta^*(T, w), x)\end{aligned}$$

Beispiel-Automat



EA RA Sprachen Sprachen von EA DEA NEA ϵ-NEA Äquivalenz

Beispiel: Rekursive Berechnung von δ^* für DEA

1. $\delta^*(0, aba)$
2. $= \delta(\delta^*(0, ab), a)$
3. $= \delta(\delta(\delta^*(0, a), b), a)$
4. $= \delta(\delta(\delta(\delta^*(0, \epsilon), a), b), a)$
5. $= \delta(\delta(\delta(0, a), b), a)$
6. $= \delta(\delta(1, b), a)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= T \\ \delta^*(T, wx) &= \delta(\delta^*(T, w), x)\end{aligned}$$

Beispiel-Automat

EA RA Sprachen Sprachen von EA DEA NEA ϵ-NEA Äquivalenz

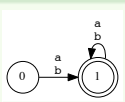
Beispiel: Rekursive Berechnung von δ^* für DEA

1. $\delta^*(0, aba)$
2. $= \delta(\delta^*(0, ab), a)$
3. $= \delta(\delta(\delta^*(0, a), b), a)$
4. $= \delta(\delta(\delta(\delta^*(0, \epsilon), a), b), a)$
5. $= \delta(\delta(\delta(0, a), b), a)$
6. $= \delta(\delta(1, b), a)$
7. $= \delta(1, a)$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= T \\ \delta^*(T, wx) &= \delta(\delta^*(T, w), x)\end{aligned}$$

Beispiel-Automat



EA RA Sprachen Sprachen von EA DEA NEA ϵ-NEA Äquivalenz

Beispiel: Rekursive Berechnung von δ^* für DEA

1. $\delta^*(0, aba)$
2. $= \delta(\delta^*(0, ab), a)$
3. $= \delta(\delta(\delta^*(0, a), b), a)$
4. $= \delta(\delta(\delta(\delta^*(0, \epsilon), a), b), a)$
5. $= \delta(\delta(\delta(0, a), b), a)$
6. $= \delta(\delta(1, b), a)$
7. $= \delta(1, a)$
8. $= 1$

Delta-Stern

$$\begin{aligned}\delta^*(T, \epsilon) &= T \\ \delta^*(T, wx) &= \delta(\delta^*(T, w), x)\end{aligned}$$

Beispiel-Automat

Abbildung 4.5: Beispielevaluation der erweiterten Übergangsfunktion für DEA

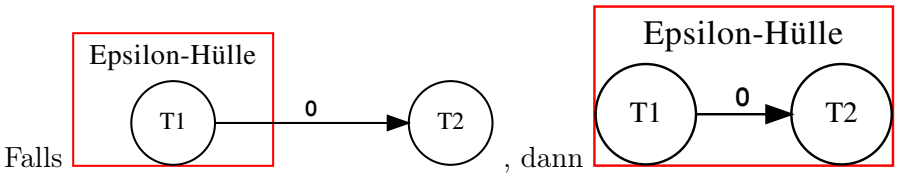


Abbildung 4.6: Rekursiver Erweiterungsschritt der Hülle

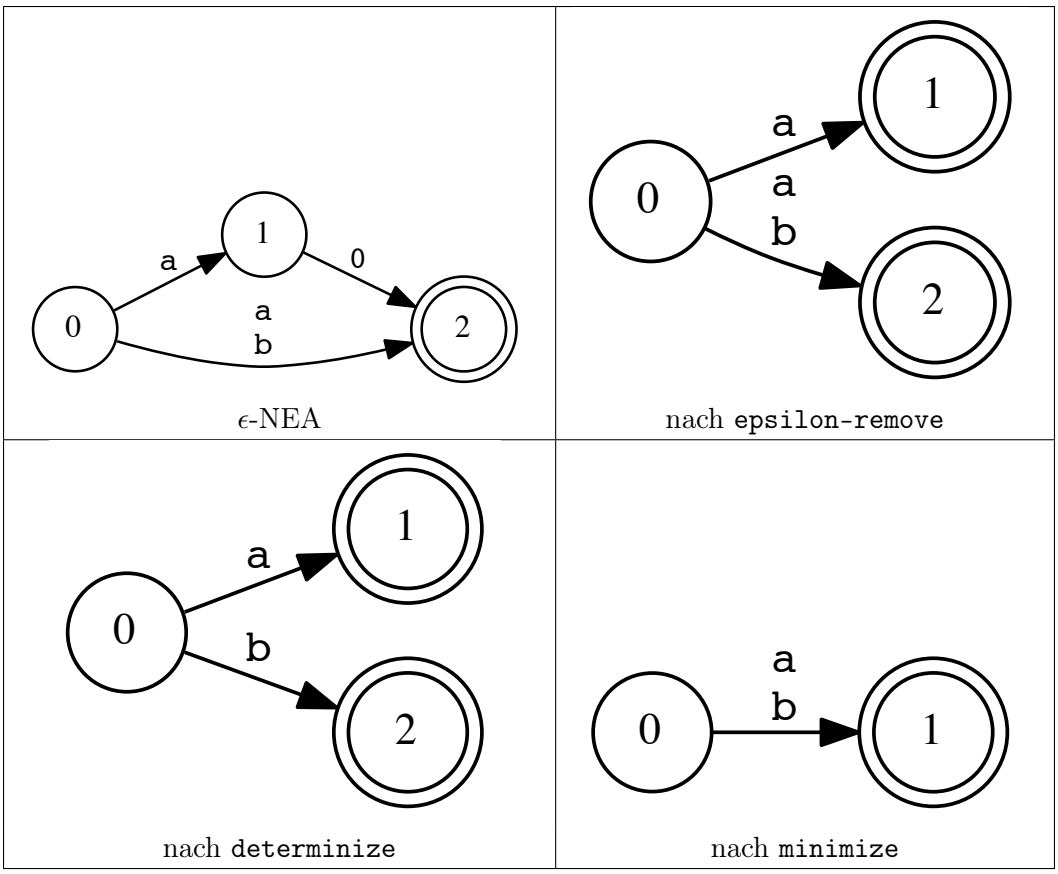
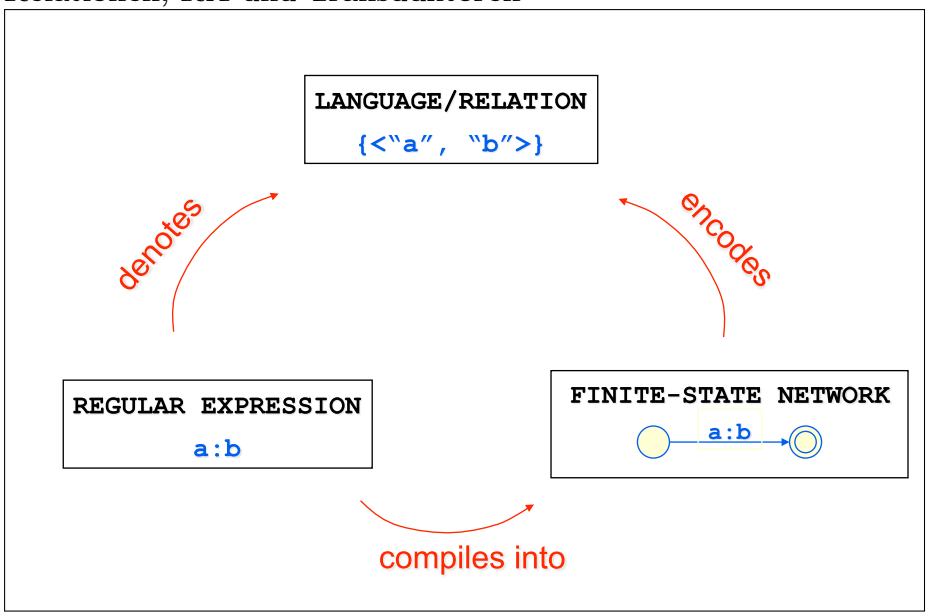


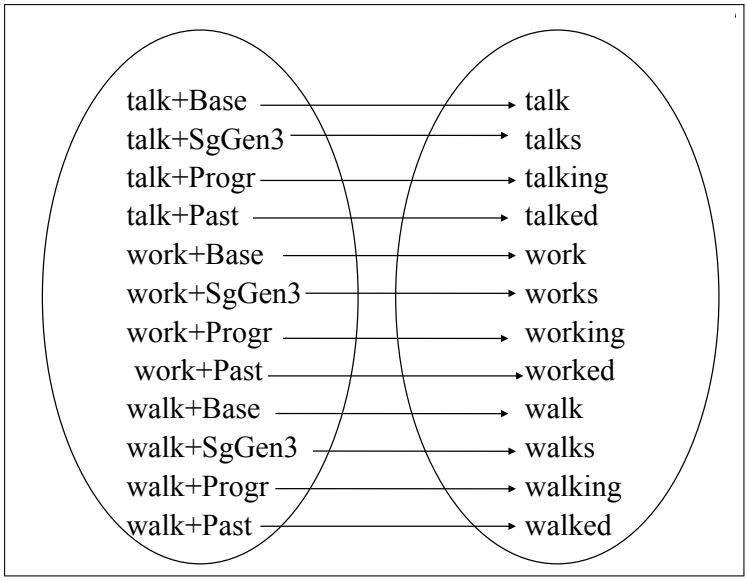
Abbildung 4.7: Effekt der 3 EA-Transformationen

Kapitel 5

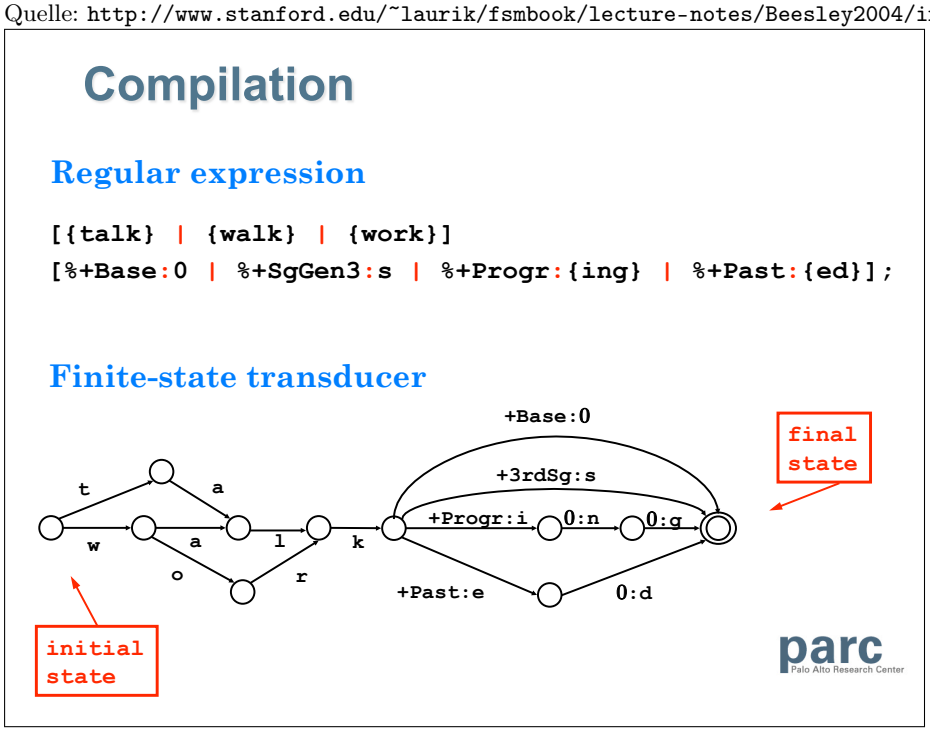
Reguläre Ausdrücke für reguläre Relationen

Relationen, RA und Transduktoren





Quelle: `http://www.stanford.edu/~laurik/fsmbook/lecture-notes/Beesley2004/index.html`



`http://www.stanford.edu/~laurik/fsmbook/LSA-207/Slides/LSA2005-Lecture1.ppt`

Relationen verknüpfen Elemente von Mengen

5.1 Formale Relationen

Geordnete Paare

Definition 5.1.1 (Geordnetes Paar)**.** Ein *geordnetes Paar* besteht aus einer ersten und einer zweiten Komponente (Koordinate). Diese werden zwischen spitzen Klammern notiert: $\langle a, b \rangle$. Oft aber auch in runden: (a, b).

Definition 5.1.2 (Gleichheit von geordneten Paaren)**.** Zwei *geordnete Paare sind gleich*, wenn sie in ihren beiden Komponenten gleich sind. Formal: $\langle a, b\rangle = \langle c, d\rangle =_{df.} a = c \wedge b = d$

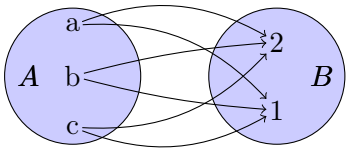
Beispiel 5.1.3 (Unterschied von geordneten Paaren und Zweier-Mengen).
Sei $a \neq b$. Dann gilt $\{a, b\} = \{b, a\}$, aber $\langle b, a\rangle = \langle a, b\rangle$ gilt nicht.

Kreuzprodukt

Definition 5.1.4 (Produktmenge, kartesisches Produkt)**.** Ein *Kreuzprodukt* zweier Mengen besteht aus der Menge der geordneten Paare, welche sich aus deren Elementen kombinieren.

$$M \times N = \{\ \langle x, y\rangle \mid x \in M \wedge y \in N\ \}$$

Beispiel 5.1.5 (Kreuzprodukt).
Sei $A = \{a, b, c\}$ und $B = \{1, 2\}$:
$A \times B = \{\langle a, 1\rangle, \langle a, 2\rangle, \langle b, 1\rangle, \langle b, 2\rangle, \langle c, 1\rangle, \langle c, 2\rangle\}$
$B \times B = \{\langle 1, 1\rangle, \langle 1, 2\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle\}$



Frage
Welche Menge ergibt sich, wenn $B = \emptyset$?

Formale Sprach-Relationen

Definition 5.1.6. Eine *formale Sprach-Relation* ist eine Teilmenge des kartesischen Produktes des Sterns von Sigma

$$R \subseteq \Sigma^* \times \Sigma^*$$

Eine Sprach-Relation ist also eine Menge von Paaren von Zeichenketten.

Beispiel 5.1.7. <+>Sei $\Sigma = \{a, b, c, A, B, C\}$.
Dann ist beispielsweise $R = \{\langle aa, AA\rangle, \langle ab, AB\rangle, \langle cc, CC\rangle\}$ eine formale Sprach-Relation über Σ.

Frage
Wie sieht ein Automat aus, der beliebige kleingeschriebene Wörter kapitalisiert?

5.1.1 Sprach-Ebenen

Obere vs. untere Sprache

$$Relation \subseteq UPPER \times LOWER$$

Definition 5.1.8 (Obere Sprache,*upper language*)**.** Die *obere Sprache* umfasst alle Zeichenketten aus der 1. Komponente des kartesischen Produkts.

Sie enthält bei lexikalischen Transduktoren traditionell die abstrakte *lexikalisch-morphologische Seite* (*lexical language*).

Definition 5.1.9 (Untere Sprache, *lower language*)**.** Die *untere Sprache* umfasst alle Zeichenketten aus der 2. Komponente des kartesischen Produkts.

Sie enthält bei lexikalischen Transduktoren traditionell die *Wortform*.

Beispiel 5.1.10 (Obere und untere Sprache im Spanischen).
$\langle$ `cantar+Verb+PresInd+1P+Sg` , `canto`$\rangle \in R_{spanisch}$

Hinweis
Bei andern Ansätzen zur Morphologie mit endlichen Automaten wie [Antworth 1990] bezeichnet man die Ebene der Wortformen mit *surface form*. Dies entspricht aber in XFST-Sprechweise der *lower language*!

5.1.2 Identitätsrelation

Identitätsrelation einer Sprache
Sie ist eine technisch wichtige, inhaltlich aber uninteressante Relation und macht Sprachen zu Relationen, ohne an der Menge der erkannten Zeichenketten etwas zu ändern.
Obere und untere Sprache der Relation sind identisch mit der zugrundeliegenden Sprache.

Definition 5.1.11 (Identitätsrelation)**.** Eine *Identitätsrelation* ID einer Sprache L über dem Alphabet Σ paart jede Zeichenkette der Sprache L nur mit sich selbst: Für alle $L \subseteq \Sigma^*$:

$$ID_L = \{\langle w, w\rangle \in \Sigma^* \times \Sigma^* \mid w \in L\}$$

Beispiel 5.1.12 (Identitätsrelation). Sei $L = \{$`canto`, `cantamos`$\}$. Dann ist
$ID_L = \{\langle canto, canto\rangle, \langle cantamos, cantamos\rangle\}$.

5.2 Endliche Transduktoren

Endliche Transduktoren (ET)

Idee der endlichen Transduktoren
Endliche Transduktoren erweitern das Konzept des endlichen Automaten, indem auf *zwei* statt auf einem Band Zeichenketten verarbeitet werden.

Unterschiedliche Interpretationen der Bänder

- Beide Bänder werden *gelesen*. Berechnete Information: Entscheidung, ob die Paare von Zeichenketten akzeptiert werden oder nicht.
- Beide Bänder werden *geschrieben*. Berechnete Information: Aufzählung der akzeptierten Paare von Zeichenketten.
- Ein Band wird *gelesen*, das andere *geschrieben*. Berechnete Information: *Aufzählung* aller möglichen Zeichenketten, welche zusammen mit den gelesenen Zeichenketten, akzeptiert werden: Analyse, Generierung, Übersetzung.

Lexikalischer Transduktor bei Wortform-Generierung
Siehe Abb. 128 auf Seite 58.

Lexikalischer Transduktor bei Wortform-Analyse
Siehe Abb. 5.2 auf Seite 58.

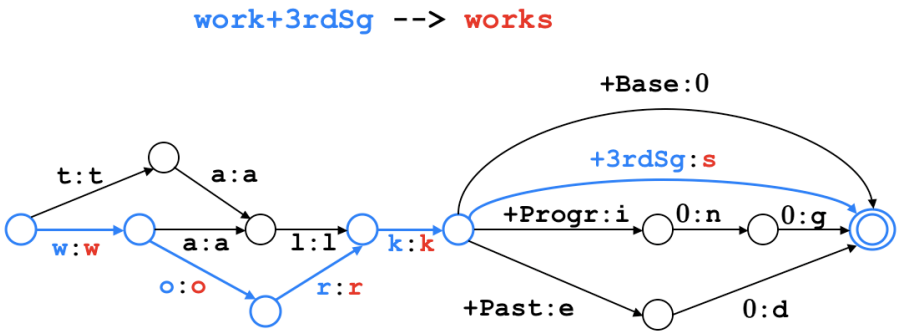


Abbildung 5.1: Lexikalischer Transduktor (von `http://www.stanford.edu/~laurik/fsmbook/LSA-207/Slides/LSA2005-Lecture1.ppt`)

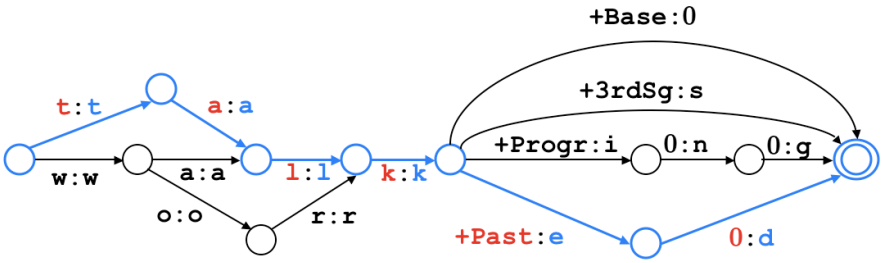


Abbildung 5.2: Lexikalischer Transduktor (von `http://www.stanford.edu/~laurik/fsmbook/LSA-207/Slides/LSA2005-Lecture1.ppt`)

5.2.1 ϵ-NET

Nicht-deterministischer endlicher Transduktor (ϵ-NET)

Definition 5.2.1 (*non-deterministic finite state transducer*)**.** Ein *nicht-deterministischer endlicher Transduktor* $T = \langle \Phi, \Sigma, \delta, S, F \rangle$ besteht aus

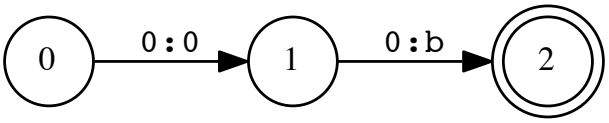
1. einer endlichen Menge *Zustände* Φ
2. einem endlichen *Alphabet* Σ
3. einer *Zustandsübergangsfunktion* $\delta : \Phi \times \Sigma_\epsilon \times \Sigma_\epsilon \to \wp(\Phi)$
4. einem *Startzustand* $S \in \Phi$
5. einer Menge von *Endzuständen* $F \subseteq \Phi$

Hinweise
Die Notation Σ_ϵ steht für $\Sigma \cup \{\epsilon\}$, wobei $\epsilon \notin \Sigma$. Ein Transduktor unterscheidet sich nur in der Übergangsfunktion von einem EA.

ET im XFST-Prolog-Format

Beispiel 5.2.2 (Direktes Einlesen von ET im PROLOG-Format).

```
xfst[0]: read prolog
        Type one or more networks in prolog format.
        Use ^D to terminate input.
arc(ex2,0,1,"0":"0").
arc(ex2,1,2,"0":"b").
final(ex2,2).
```



- Kantenrepräsentation: `arc(ID, VON, NACH, ZEICHENPAAR)`
- Symbolpaare mit identischen Symbolen können auch innerhalb eines ET als einfache Symbole geschrieben werden: `arc(ex2,0,1,"0")`.

Übergangsfunktion auf Paare von Zeichenketten

Die Bestimmung der Übergangsfunktion δ^* auf Paare von Zeichenketten setzt auf der reflexiven und transitiven Hülle der ϵ:ϵ-Übergänge auf.

Definition 5.2.3 (Epsilon-Hülle)**.** Die *ϵ-Hülle* $\delta_\epsilon^* : \Phi \to \wp(\Phi)$ eines ϵ-NET für einen Zustand $T \in \Phi$ ist rekursiv definiert durch:

- $T \in \delta_\epsilon^*(T)$
- Falls $T_1 \in \delta_\epsilon^*(T)$ und $T_2 \in \delta(T_1, \epsilon, \epsilon)$, dann $T_2 \in \delta_\epsilon^*(T)$.

Hinweis

Das Vorgehen ist analog zur Behandlung in ϵ-NEA.

Sprach-Relation von ϵ-NET

Definition 5.2.4 (Zustandsübergangsfunktion für Zeichenketten)**.** Die *auf Zeichenketten erweiterte Übergangsfunktion* $\delta^* : \Phi \times \Sigma_\epsilon^* \times \Sigma_\epsilon^* \to \wp(\Phi)$ bestimmt die Menge der erreichbaren Zustände nach dem vollständigen Lesen eines Paars von Zeichenketten.

Für alle $T \in \Phi, u, w \in \Sigma^*, x, y \in \Sigma_\epsilon$ (mit $xy \neq \epsilon$) eines ϵ-NET

$$\begin{aligned} \delta^*(T, \epsilon, \epsilon) &= \delta_\epsilon^*(T) \\ \delta^*(T, wx, uy) &= \bigcup_{T' \in \delta(\delta^*(T,w,u),x,y)} \delta_\epsilon^*(T') \end{aligned}$$

Die Sprach-Relation eines ϵ-NET ist die Menge aller Paare von Zeichenketten, für die δ^* ausgehend vom Startzustand mindestens einen Endzustand enthält.

$$L_T = \{\langle w, u\rangle \in \Sigma^* \times \Sigma^* \mid (\delta^*(S, w, u) \cap F) \neq \emptyset\}$$

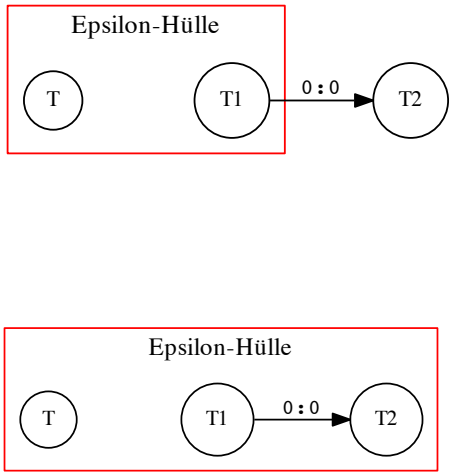


Abbildung 5.3: Rekursiver Erweiterungsschritt der ϵ-Hülle

5.2.2 Zugrundeliegender EA eines ET

Zugrundeliegender EA
Jeder ET kann als EA aufgefasst werden, der die Symbole auf den beiden Bändern des ET als Symbolpaar auf einem Band verarbeitet.

Definition 5.2.5 (zugrundeliegender EA)**.** Ein *zugrundeliegender EA* eines $ET = (\Phi, \Sigma, \delta, S, F)$ ergibt sich als $(\Phi, \Sigma', \delta', S, F)$: Für alle $a, b \in \Sigma$ und $T \in \Phi$

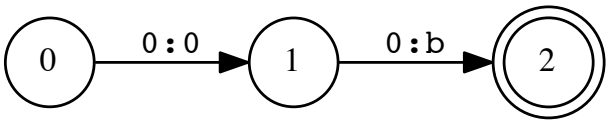
$$\begin{aligned} \Sigma' &= \Sigma \times \Sigma \\ \delta'(T, \langle a, b \rangle) &= \delta(T, a, b) \end{aligned}$$

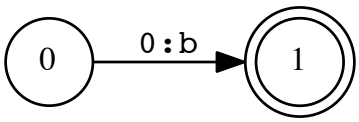
5.2.3 NET

Nicht-deterministischer endlicher Transduktor (NET)
Jeder ϵ-NET kann in einen äquivalenten NET umgewandelt werden, der keine ϵ:ϵ-Übergänge enthält, indem im zugrundeliegenden EA die entsprechende ϵ-Eliminierung gemacht wird.
Der NET kann jedoch weiterhin Übergänge vom Typ ϵ:a bzw. a:ϵ mit $a \in \Sigma$ enthalten.

Beispiel 5.2.6 (Entfernen von ϵ:ϵ-Übergängen).

```
xfst[1]: epsilon-remove net
xfst[1]: print net
Sigma: b
Flags: epsilon_free, loop_free
Arity: 2
s0 [non-det]: <0:b> -> fs1.
fs1:  (no arcs)
```

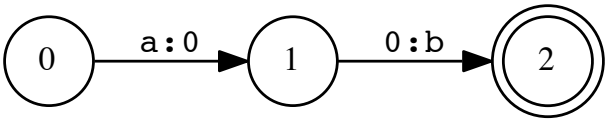




EA-Minimale Transduktoren in XFST
Das Minimieren von ET bezieht sich in XFST immer auf den zugrundeliegenden EA.

Beispiel 5.2.7 (xfst und Zustandsdiagramm).

```
xfst[0]: read regex [ a:0 0:b ] ;
xfst[1]: minimize net
xfst[1]: print flags
deterministic, pruned, minimized, epsilon_free, loop_free,
```

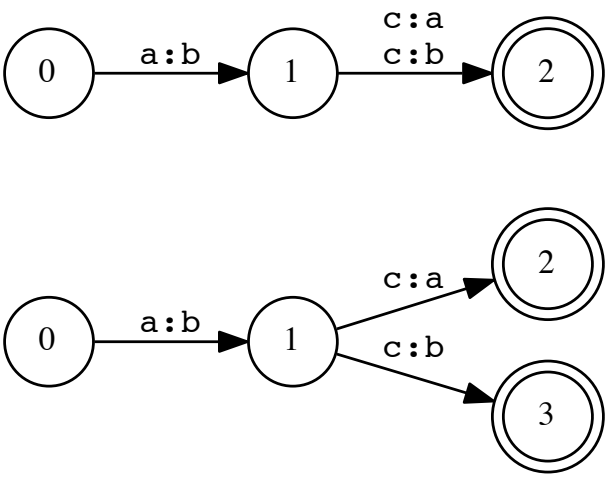


Hinweis
Die Relation kann leicht durch einen Automaten mit nur zwei Zuständen ausgedrückt werden. XFST führt eine solche Minimierung aber nicht automatisch durch. Für einfache Fälle der Form `a:0 0:b` führt der Stackbefehl `cleanup` allerdings doch noch zum erwünschten Ergebnis.

EA-Deterministische Transduktoren in XFST
Das Flag für Determiniertheit von ET bezieht sich in XFST immer auf den zugrundeliegenden EA.

Beispiel 5.2.8 (xfst und Zustandsdiagramm).

```
xfst[0]: set minimal OFF
xfst[0]: read regex a:b c:a | a:b c:b;
xfst[1]: print net
...
s0 [non-det]: <a:b> -> s1, <a:b> -> s2.
...
xfst[1]: determinize net
xfst[1]: print net
Flags: deterministic, pruned,
       epsilon_free, loop_free
```



Hinweis
EA-Deterministische Transduktoren in XFST erlauben gleiche Symbole auf *dem einen* Band, welche zu unterschiedlichen Folgezuständen führen.

5.3 Reguläre Relationen

5.3.1 Konkatenation und Stern

Konkatenation von Sprach-Relationen

Definition 5.3.1. Die *Konkatenation von Relationen* erweitert die Verkettung auf Mengen von Paaren von Zeichenketten. Für alle $R, S \subseteq \Sigma^* \times \Sigma^*$:

$$\bullet : \wp(\Sigma^* \times \Sigma^*) \times \wp(\Sigma^* \times \Sigma^*) \to \wp(\Sigma^* \times \Sigma^*)$$

$$R \bullet S = \{\langle a \bullet b, x \bullet y\rangle \mid \langle a, x\rangle \in R \wedge \langle b, y\rangle \in S\}$$

Beispiel 5.3.2. <+>Sei $R = \{\langle geb, gib\rangle, \langle geb, gab\rangle\}$ und $S = \{\langle en, st\rangle\}$.
Dann ist $R \bullet S = \{\langle geben, gibst\rangle, \langle geben, gabst\rangle\}$.

Stern einer Relation

Die Konkatenation von Relationen mit sich selbst erweitert sich wie bei den Sprachen zum Stern einer Relation.

5.3.2 Rekursive Definition

Reguläre binäre Relationen

Definition 5.3.3 (Binäre Relation)**.** Eine *Relation* über Σ heisst *regulär*, wenn sie durch folgende reguläre Mengenausdrücke beschrieben werden kann:

- Die leere Menge $\emptyset$ ist *regulär*.
- Wenn $\langle x, y\rangle \in \Sigma_\epsilon \times \Sigma_\epsilon$, dann ist die Menge $\{\langle x, y\rangle\}$ regulär.
- Wenn R_1 und R_2 regulär sind, dann auch $(R_1 \cup R_2)$.
- Wenn R_1 und R_2 regulär sind, dann auch $(R_1 \bullet R_2)$.
- Ist R regulär, dann auch R^*.

Abschlusseigenschaften regulärer Relationen

- Über der Menge der regulären Relationen sind die Operationen $R_1 \bullet R_2$, R^* und $R_1 \cup R_2$ per Definition abgeschlossen.
- Der Abschluss bezüglich der Operationen R^+, $R_1 \circ R_2$ (Komposition), R^{-1} (Inversion) lässt sich beweisen.
- Der Abschluss bezüglich der Operationen $R_1 \cap R_2$, $R_1 \setminus L_2$ und Komplementmenge $\overline{L}$ ist *nicht* gegeben.

Abschlusseigenschaften längengleicher regulärer Relationen
Reguläre Relationen, bei denen alle Paare von Zeichenketten gleich lang sind, sind abgeschlossen unter $R_1 \cap R_2$ und $R_1 \setminus R_2$.

Reguläre Relationen und ET
Die Menge der regulären Relationen ist gleich der Menge der von endlichen Transduktoren (ET) akzeptierten Sprach-Relationen.

5.4 Reguläre Ausdrücke für reguläre Relationen

5.4.1 Produkt

Das kartesische Produkt in XFST
Wenn die RA `A` und `B` die Sprachen A und B über Σ bezeichnen, dann bezeichnet `[ A .x. B ]` die Relation $A \times B$.

Beispiel 5.4.1. FrageWelche Relation beschreibt folgender Transduktor?
```
xfst[0]: read regex [{cat}|{dog}] .x. [{katze}|{hund}];
```

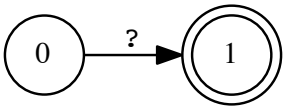
Universale Relation
Der RA `[ ?* .x. ?* ]` bezeichnet die universale Relation $\Sigma^* \times \Sigma^*$, welche irgendeine Zeichenkette mit einer beliebigen andern paart.

Alternative Notation: Doppelpunkt-Operator
Anstelle von `[ A .x. B ]` kann auch `[ A : B ]` verwendet werden. Der Doppelpunkt bindet einfach stärker. Er wird normalerweise für Symbolpaare verwendet `a:b` oder zusammen mit Klammernotation: `{cat}:{chat}`.

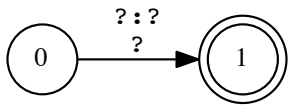
ANY-Symbol, Sprachen und Identitätsrelation
Jeder RA *`A`* für eine Sprache bezeichnet auch deren Identitätsrelation.

ANY-Symbol als Identitätsrelation
Der reguläre Ausdruck `?` steht sowohl für die Sprache aller Zeichenketten der Länge 1 (Σ^1) wie auch für die Identitätsrelation über dieser Sprache.
`?:?` bezeichnet nicht nur die Identitätsrelation, sondern auch $\Sigma^1 \times \Sigma^1$.

Beispiel 5.4.2 (ANY-Symbol in Sprache und Relation).



```
read regex [ ? ];
```




```
read regex [ ?:? ];
```

Fallstrick: UNKNOWN-Kante vs. UNKNOWN-Paar-Kante
`?:?` als Kantenbeschriftung bedeutet zwingend unterschiedliche Symbole. Nur `?` als Kantenbeschriftung erlaubt identisches Symbolpaar.

Die unterschiedliche Verwendung des Fragezeichens in regulären Ausdrücken und endlichen Automaten ist bei ?:? besonders verwirrlich zugespitzt. Als regulärer Ausdruck (ANY) schliesst es die Identität ein, aber als Kante (UNKNOWN) schliesst es sie gerade aus. ?:? ist also eigentlich ein UNKNOWN:ANOTHERUNKNOWN.

5.4.2 Inversion

Definition 5.4.3 (Inversion von Relationen)**.** Wenn `R` eine Relation darstellt, dann bezeichnet `[ R ].i` die Relation, bei der die obere und untere Sprache vertauscht sind.

Für alle Relationen `R` gilt: `R = R.i.i`.

Konstruktion eines inversen ET
Um aus einem ET einen neuen ET zu konstruieren, der die inverse Relation erkennt, müssen nur alle Symbole in den Symbolpaaren ausgetauscht werden.

5.4.3 Komposition

Operator für Komposition von Relationen
Wenn die RA `R` und `S` die Relationen R und S über Σ bezeichnen, dann bezeichnet `[ R .o. S ]` eine Relation. Sie beinhaltet ein Zeichenkettenpaar $\langle u, w\rangle$ genau dann, wenn R ein Paar $\langle u, v\rangle$ enthält und S ein Paar $\langle v, w\rangle$.

$$\{\langle u, v\rangle\} \circ \{\langle v, w\rangle\} = \{\langle u, w\rangle\}$$

Definition 5.4.4 (Relationskomposition)**.** Die *Komposition von Relationen* $R \circ S \subseteq \Sigma^* \times \Sigma^*$ ist:

$$\circ : \wp(\Sigma^* \times \Sigma^*) \times \wp(\Sigma^* \times \Sigma^*) \to \wp(\Sigma^* \times \Sigma^*)$$

$$R \circ S = \{\langle u, w\rangle \mid \exists v \in \Sigma^* (\langle u, v\rangle \in R \wedge \langle v, w\rangle \in S)\}$$

Beispiel 5.4.5 (Flexionsmorphologie mit Komposition und Ersetzung). Ein Beispiel für typisches Zusammenspiel von Komposition und Ersetzung bei der Behandlung eines Ausschnitts der regulären Verbflexion des Deutschen findet sich ►►►hier.

Beispiele zur Komposition
Welche Sprach-Relationen bezeichnen die folgenden RA?

- `[a:b] .o. [b:c]`
- `[a:b] .o. [c:b]`
- `[b:a|c:b] .o. [a:d]`
- `[c:b] .o. [a:d|b:a]`
- `[a|b|c] .o. [a|c]`

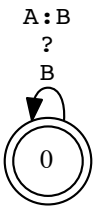
Hinweis
Die Komposition ist in XFST zusammen mit der Ersetzung ein fundamentaler Baustein und muss gründlich verstanden sein.

Hinweis
Auf EA angewendet verhält sich die Komposition identisch wie die Schnittmengenbildung:
`A .o. B` = `A & B`.

5.4.4 Ersetzung

Operator für Ersetzung (*replace*)

Beispiel 5.4.6 (xfst und Zustandsdiagramm). `read regex [ A -> B] ;`



Definition 5.4.7 (Ersetzungsoperator)**.** Wenn die RA `A` und `B` die Sprachen A und B über Σ bezeichnen, dann bezeichnet `[ A -> B ]` folgende Relation:

- Generell: Identische Paare von beliebigen Zeichenketten.
- Ausnahme: Alle Teilzeichenketten aus A in der oberen Sprache müssen mit Teilzeichenketten aus B gepaart sein.

Hinweis zur Namensgebung des Operators
Falls B genau eine Zeichenkette enthält, werden alle Vorkommen von A durch B ersetzt wie bei einer Ersetzungfunktion.

Die Semantik des Ersetzungsoperators

Beispiel 5.4.8 (Ersetzung ►►►). Gegeben sei die Relation `[ [ a | b ] -> [ c | d ] ]`. Welche Zeichenketten der unteren Sprache sind zur Zeichenkette `eab` aus der oberen Sprache gepaart?

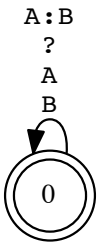
```
xfst[0]:  read regex [ [a|b] -> [c|d] ];
180 bytes. 1 state, 7 arcs, Circular.
xfst[1]: apply down eab
...
```

Reduktion des replace-Operators

`[ A -> B ] = [ NO_A [A .x. B] ]* NO_A ]`
wobei `NO_A = ~ $ [ A - 0 ]`.
Eine beliebig wiederholte Konkatenation von Zeichenketten, welche nichts aus A enthalten, mit dem Kreuzprodukt von A und B. Gefolgt von beliebigen Zeichenketten, welche ebenfalls nichts aus A enthalten.

Optionale Ersetzung

Beispiel 5.4.9 (xfst und Zustandsdiagramm). `read regex [ A (->) B ] ;`

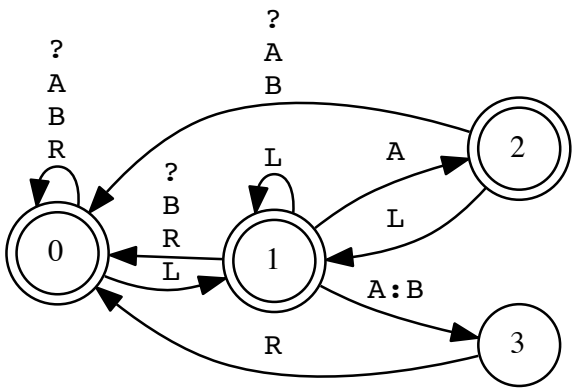


Definition 5.4.10 (Ersetzungsoperator)**.** Wenn die RA `A` und `B` die Sprachen A und B über Σ bezeichnen, dann bezeichnet `[ A -> B ]` folgende Relation:

- Generell: Identische Paare von beliebigen Zeichenketten.
- Ausnahme: Alle Teilzeichenketten aus A in der oberen Sprache können auch mit Teilzeichenketten aus B gepaart sein.

Bedingte Ersetzung (*conditional replacement*)

Beispiel 5.4.11 (xfst und Zustandsdiagramm).
`read regex [   A -> B || L _ R  ] ;`



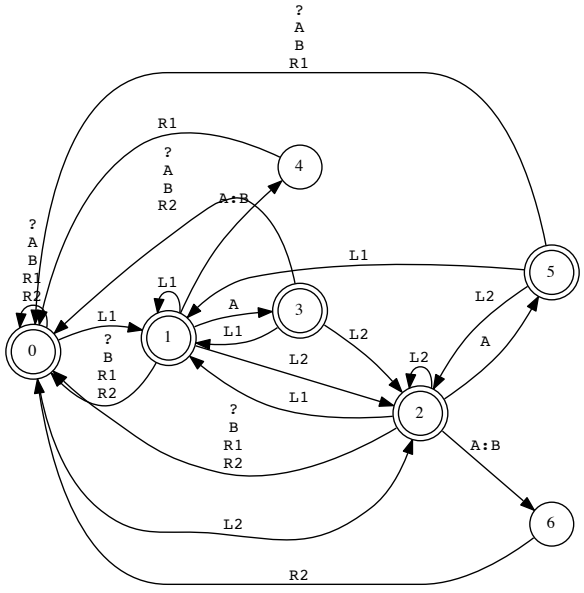
Wenn die RA `A`, `B`, `L` und `R` Sprachen über Σ bezeichnen, dann bezeichnet `[ A -> B || L _ R ]` folgende Relation:

- Generell: Identische Paare von beliebigen Zeichenketten.
- Ausnahme: Alle Teilzeichenketten aus A in der oberen Sprache müssen mit Teilzeichenketten aus B gepaart sein, falls sie nach L und vor R stehen.

Bedingte Ersetzung mit mehrfachen Kontexten
Anstelle nur eines einzigen möglichen Kontexts lassen sich beliebig viele durch Komma getrennt angeben, in denen eine Ersetzung stattfinden muss:
`[ A -> B || L1 _ R1 , L1 _ R1 , ... , Ln _ Rn ]`

Beispiel 5.4.12 (xfst und Zustandsdiagramm).

```
read regex [
    A -> B
 || L1 _ R1 , L2 _ R2
] ;
```



Wortende verankern in Kontexten

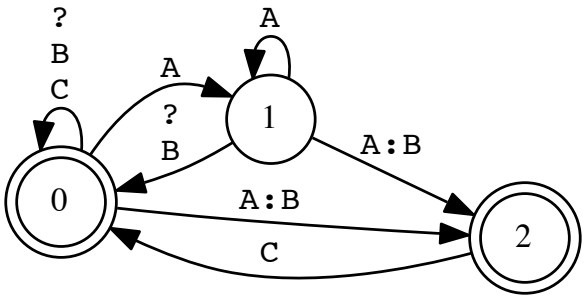
Die Spezialmarkierung für Wortanfang/-ende
Die Kontexte in `[ A -> B || L _ R , L1 _ R2 ]` sind gegen Aussen implizit mit der universalen Sprache verkettet:

```
[ A -> B || ?* L _ R ?* , ?* L1 _ R2 ?*]
```

Wie beim =>-Operator bedeutet die Spezialmarkierung `.#.` in den Kontexten Verankerung an der Wortgrenze und verhindert so die implizite Verkettung.
Im resultierenden ET ist `.#.` nicht vorhanden, es kann aber wie ein Symbol in die RA eingefügt werden.

Beispiel 5.4.13 (xfst und Zustandsdiagramm).

```
read regex [
A -> B || _ [ C | .#. ]
] ;
```



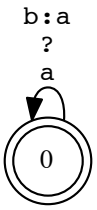
Wegen der impliziten Erweiterung dürfen Kontexte auch fehlen.

Parallele Ersetzung: Austauschen
Mehrere gleichzeitige Ersetzungen sind im Ersetzungsteil mit Komma getrennt möglich: `[ A -> B , C -> D , ..., X -> Y ]`.

Beispiel 5.4.14 (xfst und Zustandsdiagramm).

```
read regex [ a -> b, b -> a ] ;
apply up abba
baab
apply down abba
baab
```

```
read regex [a -> b] .o. [ b -> a] ;
apply up abba

apply down abba
aaaa
```



Epsilon in Ersetzung

ϵ in Kontexten: Zwecklos
Die *leere Sprache in Kontexten* macht kaum Sinn:
`[A -> B || 0 _ 0]`
= `[A -> B || ?* 0 _ 0 ?*]`
= `[A -> B || ?* _ ?*]`
= `[A -> B]`

ϵ als Ersetzung: Wichtig und nützlich
Die *leere Sprache als Ersetzung* löscht die Zeichenketten der zu ersetzenden Sprache: `[A -> 0]`

ϵ als zu Ersetzendes: Überall und endlos
Die *leere Sprache als zu Ersetzendes* fügt an beliebiger Stelle beliebig oft das zu Ersetzende ein: `[0 -> A]`
Ein solcher ET besitzt ein ϵ-Loop auf der oberen Seite. Jeder Zeichenkette der oberen Sprache entsprechen unendlich viele Zeichenketten der unteren Sprache. (Automatisch prüfbar mit `test upper-bounded` in xfst.)

Einfügen als Einmal-Ersetzung (*single insertion*)
Das einmalige Einfügen von Zeichenketten ist wichtig und nützlich.

Gepunktete Klammern (*dotted brackets*)
In Ersetzungsausdrücken `[ [. A .] -> B]` beschränken gepunktete Klammern um das zu Ersetzende die Ersetzung von ϵ in `A`. An jeder Stelle darf es nur noch einmal ersetzt werden, d.h. `B` eingefügt werden.

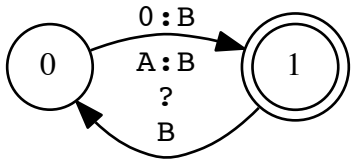
Beispiel 5.4.15 (Einmal-Einfügen).

```
xfst[0]: regex [ [. 0 .] -> "+" ];
xfst[1]: apply down xyz
+x+y+z+
xfst[1]:
```

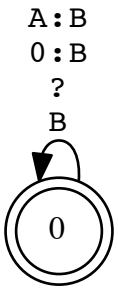
Beispiel 5.4.16 (Mehrfach-Einfügen).

```
xfst[0]: regex [ 0 -> "+" ];
xfst[1]: down xyz
Network has an epsilon loop \
on the input side.
++x+y+z
++x+y+z+
++x+yz
...
```

Einfügen als Einmal-Ersetzung (*single insertion*)
Beispiel 5.4.17 (Die Wirkung von `[. .]`).



`[ [. (A) .] -> B ]`



`[ (A) -> B ]`

Kurznotation
Der RA `[..]` wird gerne als Abkürzung für `[. 0 .]` verwendet.

ϵ-spezifische Wirkung
Die gepunkteten Klammern modifizieren die Ersetzung nur bezüglich ϵ. Die Ersetzung von nicht-leeren Teilzeichenketten besteht normal weiter.

Nicht-Determinismus in Ersetzung
Auch wenn in `[ A -> B ]` die Sprache `B` nur eine einzige Zeichenkette enthält, kann die Ersetzung nicht-deterministisch sein.

Ursache 1: Unterschiedliche Länge der zu ersetzenden Sprache
Der ET aus `[ [ a | a a ] -> b ]` bildet etwa die obere Sprache $\{aa\}$ auf $\{b, bb\}$ ab.

```
read regex {aa} .o. [[a|a a] -> b];
```

Ursache 2: Überschneidende Ersetzungen
Der ET aus `[ [ a b | b c ] -> z ]` bildet etwa die obere Sprache $\{abc\}$ auf $\{zc, az\}$ ab.

```
read regex {abc} .o. [[a b|b c] -> z];
```

Gerichtete Ersetzungsoperatoren mit Längenpräferenz

Ursache 1 des Nicht-Determinismus eliminieren
Ersetze nur die längste (`->`) oder kürzeste (`>`) Teilzeichenkette!

Ursache 2 des Nicht-Determinismus eliminieren
Ersetze nur von links nach rechts (`@...`) oder von rechts nach links (`...@`)!

Die 4 möglichen kombinierten Strategien

	von links	von rechts
lang	`A @-> B`	`A ->@ B`
kurz	`A @> B`	`A >@ B`

Siehe [Karttunen 1996] zur Implementation dieser Operatoren.

Gerichtete Ersetzungsoperatoren

Fragen

- Welche untere Sprache ergibt sich für `[ [ a | a a ] -> b ]` für die obere Sprache $\{aa\}$ mit den verschiedenen gerichteten Ersetzungsoperatoren?
- Welche für `[ [ a b | b c ] -> z ]` mit $\{abc\}$?
- Welche Konstrukte aus den regulären Suchmustern sind mit welchen gerichteten Ersetzungsoperatoren verwandt?
- Wieso ist der Operator `@->` für Tokenisierung und Chunking nützlich?

Kopierendes Ersetzen (*marking*)

Beispiel 5.4.18 (Markup mit Klammern). `[ [a|e|i|o|u] -> "[" ... "]" ]` bildet «`laut`» ab auf «`l[a][u]t`».

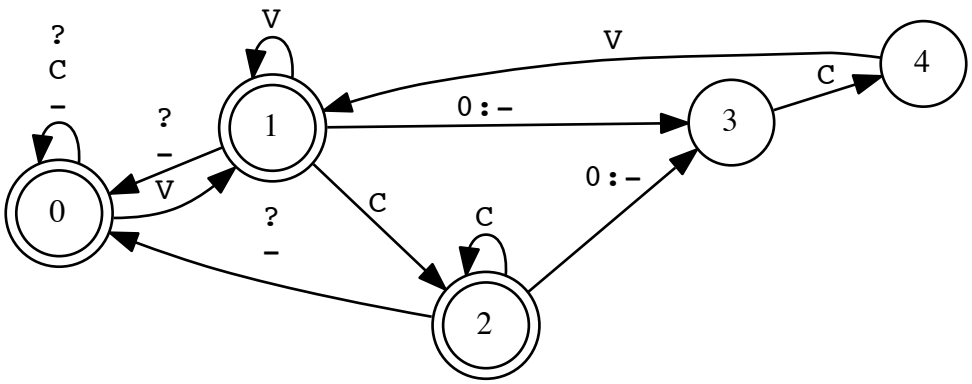
In Ersetzungsausdrücken kann `...` als Variable einmal dazu verwendet werden, die zu ersetzende Zeichenkette zu referenzieren.

Definition 5.4.19 (Marking)**.** Wenn die RA `A`, `B` und `C` die Sprachen A, B und C über Σ bezeichnen, dann bezeichnet `[ A -> B ... C ]` eine Relation.
Sie besteht aus Paaren von beliebigen Zeichenketten, welche identisch sind, ausgenommen aller Teilzeichenketten aus A in der oberen Sprache, welche gepaart sind mit einer Kopie von sich selbst, die mit einem Präfix aus B und einem Suffix aus C verkettet ist.

Beispiel: Silbifizierung ►►►

```
define C [b|c|d|f|g|j|h|k|l|m|n|p|q|r|s|t|v|w|x|z];
define V [a|e|i|o|u|y|ä|ö|ü];
define Silbifizierung [ C* V+ C* @-> ... "-" || _ C V ];

apply down silbe
apply down vorsilbe
apply down leuchtplakat
```

Beispiel 5.4.20 (Zustandsdiagramm des ET für Silbifizierung).



Ersetzungsoperatorvarianten
Neben den gezeigten Ersetzungsoperatoren gibt es noch weitere systematische Varianten

- Jede Ersetzung kann optional gemacht werden mit runden Klammern um Ersetzungspfeil.
- Jede Variante kann beliebig viele Bedingungs-Kontexte aufnehmen.
- Die Bedingungs-Kontexte können sich in beliebiger Kombination auf obere oder untere Sprache beziehen (`||` vs. \\ vs. \/ vs. `//`)
- Jede Variante kann die Ersetzung auch in der Richtung von unterer zu oberer Sprache vornehmen, indem der Pfeil umgekehrt wird (`<-` statt `->`).
- Die parallele Ersetzung erlaubt gleichzeitiges Ersetzen von Zeichenketten.

Zusammenspiel von Komposition und Ersetzung

Beispiel 5.4.21 (Der fiktionale Klassiker: `kaNpat`).

- Das abstrakte und unterspezifizierte Morphophonem `N` wird an Morphemgrenzen vor `p` als `m` realisiert.

```
[ N -> m || _ p ]
```

- Ein `p`, das einem `m` nachfolgt, wird als `m` realisiert:

```
[ p -> m || m _ ]
```

- Beachte: Jede Ersetzungsregel bildet aus Sprachen eine Sprach-Relation!
- Obige Regeln werden nacheinander als Regelkaskade angewendet: Aus `kaNpat` wird zunächst `kampat` und daraus `kammat`.
- Die Regelkaskade lässt sich durch Komposition in einen einzigen Transduktor verwandeln.

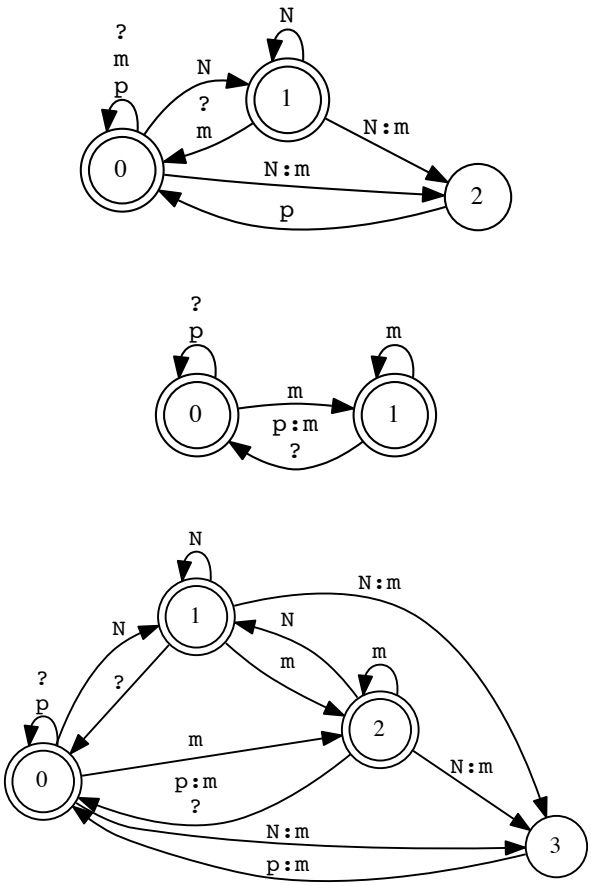


Abbildung 5.4: ET aus den kaNpat-Regeln

Regelkaskade als ein ET

XFST-Skript ►►►

```
define Rule1 [ N -> m || _ p ] ;

define Rule2 [ p -> m || m _ ] ;

read regex  Rule1 .o. Rule2 ;

apply down kaNpat

apply up kammat
```

Regelkaskade als prozedurale sequentielle Ersetzung

In PERL könnte man eine solche Kaskade leicht umsetzen mit Lookahead (`(?=...)`) bzw. Lookbehind (`(?<=...)`):

```
$_ = "kaNpat";
```

```
s/N(?=p)/m/g ;      # Rule1 [ N -> m || _ p ] ;
s/(?<=m)p/m/g ;     # Rule2 [ p -> m || m _ ] ;
```

Was kann diese PERL-Implementation nicht?

5.4.5 Projektion

Obere und untere Projektion

Definition 5.4.22 (Projektion von Relationen auf Sprachen)**.** Wenn `R` eine Relation darstellt, dann bezeichnet `[ [ R ].u ]` deren obere Sprache und `[ [ R ].l ]` deren untere Sprache.

Während das Kreuzprodukt Sprachen zu Relationen verknüpft, reduziert die Projektion Relationen zu Sprachen.

Beispiel: Dekomposition mit Ersetzung und Komposition

Cola-Maschine again
Die Cola-Maschine kann mittels Ersetzung und Komposition elegant definiert werden:
`[[D -> N^2, Q -> N^5] .o. N^5].u`

5.5 Vertiefung

Pflichtlektüre Kapitel 2 aus Beesley und Kartunnen
Gibt eine systematische Darstellung der hier vorgestellten Konzepte und Operatoren.

QUIZ Leichtes QUIZ zur Komposition

QUIZ Ersetzung und Komposition in Kombination Leider ist die Theorie der regulären Relationen oft weniger einheitlich und einfach dargestellt, als dies bei den regulären Sprachen der Fall ist.
Eine formalere Einführung in die Theorie der regulären Relationen als [BEESLEY und KARTTUNEN 2003b] gibt [KAPLAN und KAY 1994]
Die Kompilation des *replace*-Operators und seiner Varianten wird in [KARTTUNEN 1995] beschrieben – durchaus ein lesenswerter Artikel.
Wem die mengentheoretischen Formalisierungen Probleme bereiten, soll das "Formale Propädeutikum II" `http://www.cl.uzh.ch/siclemat/lehre/form-prop` oder ein entsprechende Einführung (z.B. Carstensen et al.) studieren.

Kapitel 6

XFST-Beispiele: Ersetzung und Komposition

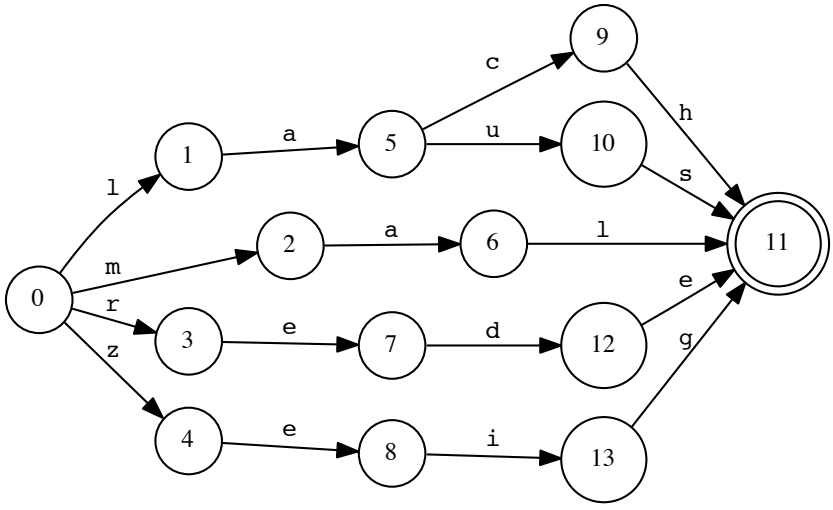
6.1 Regelmässige Verben

Flexionsmorphologie à la "Realizational Morphology"

- [Karttunen 2003] hat den Ansatz von [Stumpf 2001] mittels Finite-State-Methoden implementiert für Lingala (afrikanische Verkehrssprache).
- Hier: eine durchaus nicht perfekt Umsetzung für deutsche Verben[1]

Komponente 1: Stämme ►►►

```
define Stems [ {lach}|{rede}|{mal}|{zeig}|{laus} ];
```



- Was fällt auf bei den Stämmen?

[1] `http://coli.lili.uni-bielefeld.de/~metzing/ws0506/xfst/xfst_verb.xfst`

Komponente 2: Definition der lexikalischen Grammatik
Welche Werte für welche Merkmale? Was verbindet Merkmalwertpaare?

```
define Person [ Per ":" [ 1|2|3 ] ];

define Number [ Num ":" [ Sg|Pl ] ];

define Tense  [ Tns ":" [ Past|Pres ] ];
```

Wie werden die Merkmale geordnet? Welche Grenzmarkierungen gibt es?

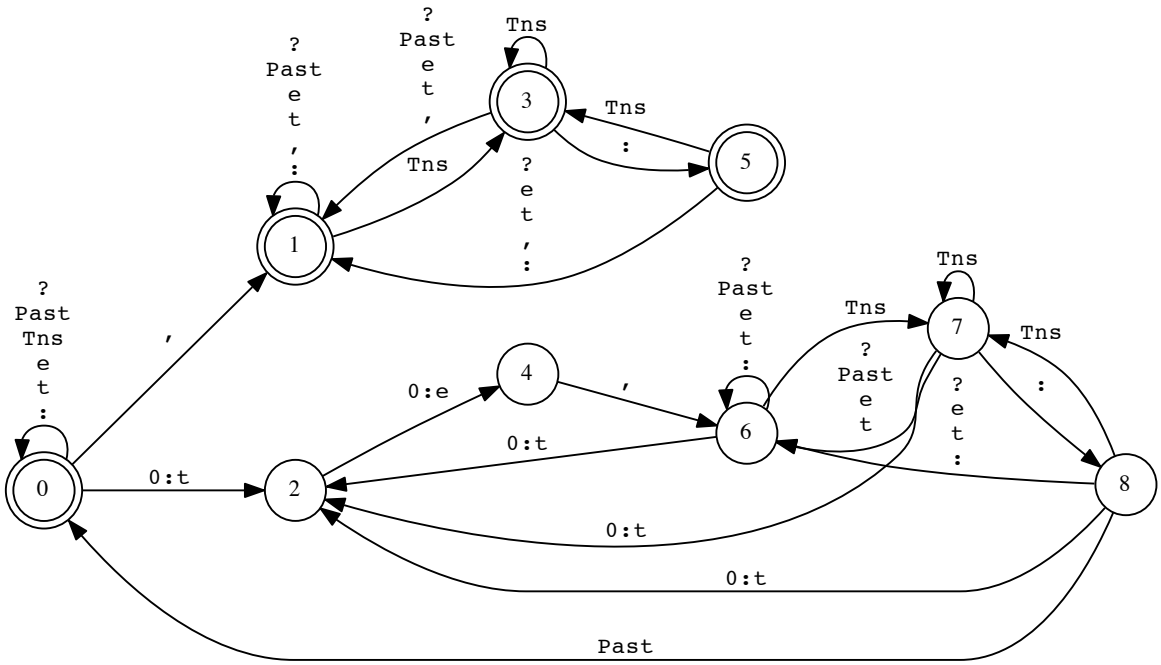
```
define Features  [ Person " " Number " " Tense ];

define VerbLex [ Stems "," Features ];
```

Komponente 3: Regel für Imperfekt
Hänge ein -**te** (an den Stamm), falls das Merkmal `Tns` den Wert `Past` hat. Beispiel: sag → sag-**te** ; rede → rede-**te** ; laus → laus-**te**

```
#Rule Block 1
define R101 [ [..] -> {te} || _ "," [ $[Tns ":" Past] ] ];
define RB1 [ R101 ];
```

- Warum wird `[..]` verwendet und nicht `0`?
- Wird hier "an den Stamm" angehängt?
- Wieviele Merkmalwertpaare `Tns ":" Past` dürfen bzw. können überhaupt links oder rechts vom Kontext vorkommen?

Transduktor für Imperfektregel



Komponente 3: Regeln für Personalendungen
Hänge die Personalendung an, welche Person und Numerus entspricht!
Beispiel: sag → sag-**st** ; rede-te → mal-te-**t**

```
#Rule Block 2 - Person/Number
define R201 [ [..] -> {e}  || _ "," [ $[Per ":" 1 " " Num ":" Sg]] ];
define R202 [ [..] -> {st} || _ "," [ $[Per ":" 2 " " Num ":" Sg]] ];
define R203 [ [..] -> {t}  || _ "," [ $[Per ":" 3 " " Num ":" Sg]] ];
define R204 [ [..] -> {en} || _ "," [ $[Per ":" 1 " " Num ":" Pl]] ];
define R205 [ [..] -> {t}  || _ "," [ $[Per ":" 2 " " Num ":" Pl]] ];
define R206 [ [..] -> {en} || _ "," [ $[Per ":" 3 " " Num ":" Pl]] ];

define RB2 [ R201 .o. R202 .o. R203 .o. R204 .o. R205 .o. R206 ];
```

- Könnte man dasselbe mit weniger Regeln oder kürzer schreiben?
- Wieviele Zustände hat wohl `RB2`?
- Sind die Endungen linguistisch korrekt?

Komponente 4: Korrektur von Übergeneralisierung
Beispiel: rede-~~ø~~t; mal-te-~~t~~

- Was passiert, wenn man Stämme wie `beehr` oder `bet` nimmt?

```
# Im Imperfekt hat 3. Person kein -t am Wortende.
define M01 [ {tet} -> {te} || _ [$[Per":"3" "Num":"Sg" "Tns":"Past]] ] ;
```

```
# In Stämmen mit e am Schluss fällt das e in der Personalendung weg.
define M02 [ {ee} -> {e} ];

define MP [ M01 .o. M02 ];
```

Vor-Montage: Zusammenfügen von Stämmen und Regeln

```
# Verbtransduktor mit lexikalischer Information auf Wortformebene
define VerbNC [ VerbLex .o. RB1 .o. RB2 .o. MP ];



xfst[0]: read regex VerbNC ;
xfst[1]: random-lower 5
zeigten,Per:3 Num:Pl Tns:Past
lachen,Per:3 Num:Pl Tns:Pres
lachtest,Per:2 Num:Sg Tns:Past
hoffen,Per:1 Num:Pl Tns:Pres
zeigte,Per:1 Num:Sg Tns:Past
```

- Die untere Sprache enthält nebst den Wortformen noch alle morphologischen Merkmale!
- Wie können wir alles ausser den Wortformen entfernen auf der unteren Sprache?

Komponente 5: Wortformen ausfiltern

```
# Zulässige Zeichen auf der Wortformen-Ebene

define L [ a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|ä|ö|ü|ß ];

#
define Cleanup [ \L -> 0 ];

# Bereinige die untere Sprache des Transduktors!
define Verb [ VerbNC .o. Cleanup ];
```

Fragen

- Was macht der Transduktor namens `Cleanup` (in Prosa ausgedeutscht)?
- Warum hat man die morphologischen Merkmale überhaupt auf der unteren Sprache gehabt?
- Sind alle erzeugten Wortformen korrekt?

6.2 Chunking

Einfache Chunking-Regel

Chunking mit Longest-Matching-Strategie
[Beesley und Karttunen 2003b, 74] geben eine einfache Regel zur NP-Erkennung (d=Det; a= Adj; n=Noun; v=Verb):

```
read regex [ (d) a* n+ @-> %[ ... %] ] ;
xfst[1]: down dannvaan
[dann]v[aan]
```

Problem
Wie kann man aus dieser Idee einen richtigen Chunker machen?

- Definition eines generellen Input- und Output-Formats
- Benutzung von `lookup`-Tool

Input für getaggten Text im Penn-Treebank-Format

Input-Klammerformat: (POS-TAG SPACE WORD)
(JJ U.K.)(NN base)(NNS rates)(VBP are)(IN at)(PRP$ their)(JJS highest)(NN level)(IN in)(CD eight)(NNS years)(. .)

Output-Klammerformat mit zusätzlicher Klammerung
(NP (JJ U.K.)(NN base)(NNS rates))(VBP are)(IN at) (NP (PRP$ their)(JJS highest)(NN level))(IN in) (NP (CD eight)(NNS years))(. .)

Schützen von wörtlichen Klammern für korrekte Verarbeitung
Schütze alle Klammern, welche als Interpunktionstoken oder als Teil von PoS-Tags vorkommen. Konvention im Penn-Treebank-Format:

- "-RRB-" für ")"
- "-LRB-" für "("

POS-Tags, Wörter und Input-Sprache ►►►
Definition eines generellen Input- und Output-Formats

```
# Penn-Treebank-Tags
define PTBT ["#" | "$" | "’’" | "-LRB-" | "-RRB-" | "," | "." | ":" |
  "CC" | "CD" | "DT" | "EX" | "FW" | "IN" | "JJ" | "JJR" |
  "JJS" | "MD" | "NN" | "NNP" | "NNPS" | "NNS" | "PDT" |
  "POS" | "PRP" | "PRP$" | "RB" | "RBR" | "RBS" | "RP" |
  "SYM" | "TO" | "UH" | "VB" | "VBD" | "VBG" | "VBN" |
  "VBP" | "VBZ" | "WDT" | "WP" | "WP$" | "WRB" | "‘‘" ] ;

# Mögliche Wörter (mit vorangehendem Leerzeichen)
define WORD [ " " [ ?* - [$ [" " | "(" | ")"]]] ] ;

# Input-Sprache
define INPUT [ "(" PTBT WORD ")" ]+;
```

NP-Chunk-Regel forfsmieren

DET, ADJ und NOUN im Penn-Treebank-Format

```
define DET [ "(" "DT" WORD ")" ];

define ADJ [ "(" ["CD"|"JJ"|"JJR"|"JJS"] WORD ")" ];

define NOUN [ "(" ["NN"|"NNS"|"NNP"|"NNPS"] WORD ")"];
```

Chunking-Regel im Penn-Treebank-Format

```
define RuleNP [
     [ ( DET ) ADJ* NOUN+ @-> "(" "NP " ... ")" ]
];
```

- Die obere Sprache ist der getaggte Text.
- Die untere Sprache ist der gechunkte Text.

Chunking

```
# lookup-tool liest lower-language, deshalb Inversion
read regex  [INPUT .o. RuleNP].i ;

# Transduktor in binärem Format speichern
save stack RuleNP.fst;

# Aufruf aus xfst mit optimierten Optionen für lange Wörter
# und deterministischem Output
system lookup RuleNP.fst -flags "v1cxmbLLTTI10000" < conll00-penn-tag.txt \
       > conll00-penn-chunk.txt
system head -n 10 conll00-penn-chunk.txt
```

- Könnte man auch noch PP- und Verbal-Chunks einbauen?

Automatische Evaluation des Chunkers

Um die Qualität des Chunkers quantifizieren zu können, evaluiere automatisch gegenüber einem sog. Gold-Standard.

```
! Automatische Evaluation
system lookup RuleNP.fst -flags "v1xmbLLTTI10000" < conll00-penn-tag.txt \
       | penn-chk-to-chk.perl -c3 > result.txt
system lam conll00-gold.chk -s ' ' result.txt | conlleval.perl
```

- `penn-chk-to-chk.perl` konvertiert das Output-Format in das IOB-Format, welches das Evaluationsskript `conlleval.perl` benötigt.
- `conll00-gold.chk` ist ein Gold-Standard aus der Shared-Task der CoNLL-Konferenz 2000: `http://www.cnts.ua.ac.be/conll2000`
- Chunker, getaggter Input und Gold-Standard sind als Zip-Archiv verfügbar: `http://www.cl.uzh.ch/siclemat/lehre/fs13/fsm/script/demo/xfst2/ptb-chunker.zip`

Resultate der Chunking-Shared-Task der CoNLL 2000

test data	precision	recall	$F_{\beta=1}$
Kudoh and Matsumoto	93.45%	93.51%	93.48
Van Halteren	93.13%	93.51%	93.32
Tjong Kim Sang	94.04%	91.00%	92.50
Zhou, Tey and Su	91.99%	92.25%	92.12
Déjean	91.87%	91.31%	92.09
Koeling	92.08%	91.86%	91.97
Osborne	91.65%	92.23%	91.94
Veenstra and Van den Bosch	91.05%	92.03%	91.54
Pla, Molina and Prieto	90.63%	89.65%	90.14
Johansson	86.24%	88.25%	87.23
Vilain and Day	88.82%	82.91%	85.76
baseline	72.58%	82.14%	77.07

Abbildung 6.1: Resultate der Chunking-Shared-Task der CoNLL 2000 [Tjong Kim Sang und Buchholz 2000, 131]

Vertiefung

- Penn-Treebank-Tagset im CLab: `http://www.cl.uzh.ch/clab/hilfe/ptts`
- Die verwendeten Befehlszeilenwerkzeuge sind alle auf `kitt.cl.uzh.ch` installiert und direkt via `xfst` verwendbar.
- Information zum Sprachmodell des Goldstandards und zu den Resultaten der Shared-Task enthält [Tjong Kim Sang und Buchholz 2000]
- [Grefenstette 1996] enthält eine gute Beschreibung zum Partial Parsing (Chunking mit partieller Satzgliederkennung) mit Hilfe von xfst-Werkzeugen.

6.3 Tokenisierung

Covingtons Tokenizer in XFST ▶▶▶

```
define AlphaNumeric [A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|a|b|c|d|e
   |f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|%0|1|2|3|4|5|6|7|8|9] ;

define WS ["\n"|" "|"\t"] ;
define WORD [ AlphaNumeric + ] ;
define SYMBOL [ \ Alphanumeric ] ;

define TOKEN [ WORD|SYMBOL ] ;

define Downcase [A->a, B->b, C->c, D->d, E->e, F->f, G->g, H->h, I->i, J->j,
   K->k, L->l, M->m, N->n, O->o, P->p, Q->q, R->r, S->s, T->t, U->u, V->v, W->w,
   X->x, Y->y, Z->z] ;

define NormalizeSpace [ WS+ @-> "\n"] ;
```

```
define Tokenizer [
   [ TOKEN  @->  ... "\n" ]
   .o.  NormalizeSpace
   .o.  Downcase ];
```

Covingtons Tokenizer in XFST mit Funktionssymbolen ►►►
Die neue Version hat eingebaute Listen von Zeichen und Funktionen

```
# xfst[0]: write definitions
# xfst[0]: print lists
# Dokumentation zu neuen Funktionen
# http://www.stanford.edu/~laurik/.book2software/novelties.pdf

define WS [ [ cntrl | space ]+ ];
define WORD [ alnum + ] ;
define SYMBOL [ \ [WORD|WS] ] ;

define TOKEN [ WORD|SYMBOL ] ;

define Tokenizer [ [ TOKEN  @-> ... "\n" ] .o. [ WS @-> "\n" ] ];

define Normalizer DownCase(Tokenizer, L);
# Function zum Downcasen auf Lower-Side
```

Befehlszeilentool `tokenize`

Tool `tokenize` **vs.** `lookup`

- Beide Tools lesen Zeichenketten der unteren Sprache ein und berechnen die damit gepaarten Zeichenketten der oberen Sprache wie bei `apply up`.
- `lookup` verarbeitet immer zeilenweise (1 Token pro Zeile).
- `lookup` berechnet nicht-deterministisch alle damit gepaarten Zeichenketten der oberen Sprache.
- `tokenize` liest beliebigen Eingabestrom von Zeichen und benötigt Transduktoren mit `?*` als untere Sprache.
- `tokenize` benötigt Transduktoren, welche deterministisch auf die obere Sprache abbilden.

Wie beim Chunker ist bei der Montage obere und untere Sprache noch vertauscht und muss beim Abspeichern noch invertiert werden.

Schillers Tokenizer mit Abkürzungsbehandlung ►►►

```
define WS [" "|"\t"|"\n" ];
define SYMBOL [ "\""|"."|","|";"|{''}|{...} ] ; ! noch mehr

define WORD [ \ [ WS | SYMBOL ] ] + ;
define ABBR [ {A.} | {Mr.} | {Mrs.} | {E.g.} ] ; ! noch mehr
```

```
define DIGIT [ "0"|1|2|3|4|5|6|7|8|9 ] ;
define NUMSYMBOL [ "-"|"."|"," ] ;
define NUM [ [ DIGIT + ] / NUMSYMBOL ] ; ! /=Ignoring-Op

define TOKEN [ SYMBOL | WORD | ABBR | NUM ] ;

define Tokenizer [
   [ TOKEN @->  ... "\n" ]
   .o.  [ WS+ @->  "\n" ]];

read regex Tokenizer.i ;
save stack tok2.fst
system echo 'E.g. 120.000 New Yorker live in New York.'| tokenize tok2.fst
```

Schillers Tokenizer mit naiver fsmtiword-Behandlung ►►►

```
source tok2.xfst

define MWT [ {New York} | {Ad hoc} ] ;

define TOKEN [ TOKEN | MWT ] ;
define WS1 [ WS+ & [ $ "\n" ] ] ;

define Tokenizer [
   [ TOKEN @->  ... "\n"]
   .o. [ WS1 @->  "\n" ]
 ];

read regex Tokenizer.i ;
save stack tok3.fst
system echo 'E.g. 120.000 New Yorker live in New York.' | tokenize tok3.fst
```

Frage
Wo liegt das Problem?

Schillers Tokenizer mit fsmtiword-Behandlung ►►►

```
source tok2.xfst

define MWT  [ {New York} | {Ad hoc} | {to and fro}] ;

define TOKEN [ TOKEN| " " MWT " "] ; ! WS-Markierung bei MWT
define BOUND [ SYMBOL | WS | .#. ] ;

define WS1 [ WS+ & [ $ "\n" ]] ;

define Tokenizer [
[WS+ @-> " "]
```

```
.o. [ MWT @-> " " ... " " || BOUND _ BOUND ]
 .o. [TOKEN @-> ... "\n" ]
  .o. [ WS1 @-> "\n" ]
] ;

read regex Tokenizer.i ;
save stack tok4.fst
system echo 'E.g. 120.000 New Yorker live in New York.' | tokenize tok4.fst
```

Kapitel 7

Entstehung der Finite-State-Morphologie

Lernziele

- Kenntnis der Ursprünge und einiger wichtiger Meilensteine in der Endlichen-Automaten-Morphologie
- Einblick, über welche (Um-)wege sich wissenschaftliche Erkenntnisse (und Modeströmungen) durchsetzen können

7.1 Vorgeschichte

Regelbasierte generative Phonologie (Chomsky und Halle)

Der "Sound-Pattern-Of-English"-Ansatz (SPE) von 1968

- [Chomsky und Halle 1968] formalisieren die traditionelle Phonologie als geordnete Folge von Ersetzungsregeln (*rewrite rules*):
- $\alpha \rightarrow \beta / \gamma \_ \delta$
  "Ersetze α durch β im Kontext von γ und δ."
- Als Kontexte dienen Zeichenketten oder Merkmal(wert)strukturen.
- Generierungsperspektive: Die abstrakte phonologische Form wird über Zwischenrepräsentationen in die Oberflächenform überführt.
- Solche Regeln sehen aus wie kontext-sensitive Grammatikregeln, welche mächtiger sind als Grammatikregeln für kontextfreie oder reguläre Sprachen.

Regelbasierte Phonologie: Auslautverhärtung

Definition 7.1.1. *Auslautverhärtung* ist das Stimmloswerden stimmhafter Obstruenten im Silbenauslaut, d.h. insbesondere auch am Wortende.
Im Deutschen sind beispielsweise die Plosive /b d g/ betroffen.

Beispiel 7.1.2 (Auslautverhärtung im Deutschen).

- 'b' wird als 'p' ausgesprochen in "Dieb" oder "Diebstahl".

- 'b' wird stimmhaft ausgesprochen in "Diebe" oder "Diebestour".
- Produktive phonologische Regel: $b \rightarrow p/ \_ \#$

- $\#$ steht hier für Silbengrenze.

Douglas C. Johnsons unbeachtete Erkenntnis

Die Verwendung der phonologischen Ersetzungsregeln

- 1972 zeigt [Johnson 1972] in seiner Dissertation "Formal Aspects of Phonological Description": Phonologen machen mit den Ersetzungsregeln typischerweise keine echt kontextsensitiven Sprachbeschreibungen.
- Phonologen haben β in $\alpha \rightarrow \beta/\gamma \_ \delta$ nie rekursiv ersetzt durch dieselbe Regel. D.H. nach der Ersetzung von α zu β in $\gamma\alpha\delta$ zu $\gamma\beta\delta$ wurde damit höchstens noch in γ oder δ ersetzt.
- Diese Beschränkung erlaubt die Modellierung von phonologischen Ersetzungsregeln mittels regulärer Relationen, d.h. Transduktoren.
- Johnson kennt die mathematischen Ergebnisse von [Schützenberger 1961], dass sich die sequentielle Anwendung von Transduktoren durch 1 komponierten Transduktoren ausdrücken lässt.

7.2 Geschichte

Der Beginn der Endlichen-Automaten-Morphologie (EAM)

Die "Gründerväter": K^4

- 1981 treffen sich Kimmo Koskenniemi, Lauri Karttunen, Martin Kay und Ronald M. Kaplan an einer Konferenz in Austin und entdecken ihr gegenseitiges Interesse an morphologischer Analyse.
- Kay und Kaplan haben bereits 1980 die Erkenntnisse von Johnson 1972 erneut entdeckt.
- 1983 publiziert Koskenniemi seine Dissertation "Two-level morphology: A general computational model for word-form recognition and production", welche die 3 wesentlichen Konzepte der EAM enthält:
  Sprachunabhängigkeit, formalisiertes Modell, Bidirektionalität (Generierung und Analyse).
- Wichtig: Keine sprachspezifischen Ad-Hoc-Cut-and-Paste-Programme!

Compiler für EA-Morphologie

Von der Handkompilation zu automatischen Kompilation

- Das System von Koskenniemi verwendete handgeschriebene Transduktoren, welche parallel arbeiteten und deren Erstellung äusserst mühsam und anspruchsvoll war. Viele weitere Implementationen von Zwei-Ebenen-Morphologie-Systemen folgten.

- 1985 entwickelten Koskenniemi und Karttunen einen Kompiler in LISP, der die Zwei-Ebenen-Regeln automatisch in Transduktoren übersetzte (unter Benutzung von EA-Implementationen von Kaplan).
- 1991 entstand der TWOLC-Kompiler (in C), mit dessen Hilfe industrielle Morphologiesysteme für zahlreiche Sprachen entstanden (Xerox).
- Ab 1995 stand mit dem Replace-Operator eine direkte Umsetzung der sequentiellen Ersetzung zur Verfügung.

Einfluss der EA-Morphologie in der Linguistik

- Im Prinzip existiert eine starke Verbindung zum SPE-Modell.
- Aber: Zwei-Ebenen-Morphologie erlaubt oberflächlich betrachtet nicht die Mächtigkeit der Sequentialität, was lange nicht geschätzt wurde.
- Der Aufstieg der Optimalitätstheorie (OT) in den 90-ern in der theoretischen Morphologie geht einher mit einer Abwertung des SPE-Modells (insbesondere der Sequentialität).
- Die OT selbst hat viele Berührungspunkte mit EA-Techniken und kann teilweise direkt implementiert werden mit EA-Werkzeugen.
- Zu diesem Zeitpunkt wurde in der CL mit dem Ersetzungsoperator das sequentielle Modell paradoxerweise wieder attraktiv.

7.3 Gegenwart und Zukunft

Ausblick

Technische Weiterentwicklung
Eine wichtige Entwicklung geht in Richtung Transduktoren mit gewichteten Übergängen. Pionierarbeit dazu wurde von [Mohri 1997] im Kontext der Verwendung von Transduktoren zur Spracherkennung in den 90-er-Jahren geleistet.
Die Finite-State-Werkzeuge, welche der tagh-Morphologie `http://www.tagh.de` zugrundeliegen, erlauben Gewichte an den Übergängen.

Theoretische Weiterentwicklung
Lassen sich neue Theorien wie bei der OT und der *Realizational Morphology* mit EA-Techniken mit theorienahen Formalismen implementieren?

7.4 Vertiefung

- Der Artikel [Karttunen und Beesley 2005] beschreibt in kompakter Form die Entwicklung der EA-Morphologie (natürlich mit einem nicht ganz neutralen Schwerpunkt und Horizont).
- Wer interessiert ist, wie Patente im Bereich der EA-Technik aussehen können, kann das amerikanische Patent von [Mohri 2000] für AT&T anschauen. Nebst dem wissenschaftlichen Publizieren kann das Erhalten von Patenten eine weitere Betätigung sein ...

Kapitel 8

Klassische 2-Ebenen-Morphologie: lexc und twol

Lernziele

- Kenntnis über die wichtigen Konzepte der Zwei-Ebenen-Morphologie: morphologische (Repräsentations-)Ebenen, Morphotaktik, phonologische/orthographische Alternation, Analyse und Generierung (Bidirektionalität)
- Kenntnis über die Strukturierung von Lexika mit Hilfe von Fortsetzungsklassen und deren Implementation in `lexc`
- Kenntnis über Zwei-Ebenen-Regeln, ihre parallele Anwendung und die Verwendung von `twolc` zur Regelkompilation
- Kenntnis über GERTWOL, ein industrielles Morphologieanalyseprogramm, das aus Ressourcen eines gedruckten Wörterbuches entstanden ist.

8.1 Einleitung

8.1.1 Ebenen

Zwei Ebenen

Beispiel 8.1.1 (GERTWOL-Morphologieanalysesystem).

```
Qualitätsmäuse
        "*qual~ität\s#maus"  S FEM PL NOM
        "*qual~ität\s#maus"  S FEM PL AKK
        "*qual~ität\s#maus"  S FEM PL GEN
```

Ebene der Wortform (*surface form*)
Die konkrete *orthographische* oder *phonetische* Form eines Wortes.

Lexikalische Ebene (*lexical form*)
Die abstrakte Grundform mit *kategorialen, morphosyntaktischen, morphotaktischen* und weiteren *lexikalischen* Informationen.

Beispiel: Informationen auf der lexikalischen Ebene

Bei GERTWOL

```
"*qual~ität\s#maus" S FEM PL AKK
```

- *Wortartenkategorie*: `S` = Substantiv
- *Morphosyntaktische Merkmale*: `FEM` = Feminines Genus, `PL` = Numerus Plural, `AKK` = Kasus Akkusativ
- *Morphotaktische Gliederung*: `#` = Kompositionsgrenze, `\` = Grenze vor Fugenelement, `~` = Derivationsgrenze
- *Orthographische Information*: `*` = Grossschreibung des nächsten Buchstabens
- *Morpheme*: `qual` `ität` `s` `maus` = Graphematische Darstellung aller Morph(em)e des Stamms (d.h. Flexionselemente fallen weg)

Demo und weitere Informationen zu den Informationen der lexikalischen Ebene unter `http://www2.lingsoft.fi/doc/gertwol/intro/`

Die Verknüpfung der zwei Ebenen

Verknüpfung

Die Zwei-Ebenen-Morphologie modelliert die Verknüpfung der beiden Ebenen durch endliche Transduktoren, d.h. als reguläre Relation.

Repräsentation der Ebenen

Die beiden Ebenen werden damit durch die obere und untere Sprache eines ET repräsentiert.

Abbildung 8.1: Die Zuordnung der beiden Ebenen durch einen Pfad eines ET

Bidirektionalität

Zwei-Ebenen-Morphologie erlaubt Analyse wie Generierung. In beiden Richtungen kann *eine* Eingabe *mehrere* Ausgaben erzeugen.

Dekomposition von Lexikon

Komplexität und ihre Reduktion

Wieso komplex? Es geht um die Verwaltung von Tausenden von Morphemen und ihrer zulässigen Kombinationen mitsamt ihren lexikalischen Merkmalen auf der lexikalischen Ebene gepaart mit den entsprechenden Wortformen.

Dekomposition der lexikalischen Ebene (Lexikon)

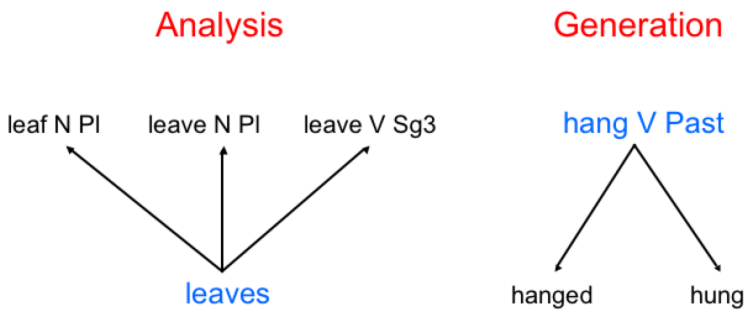


Abbildung 8.2: Mehrdeutigkeit von Analyse und Generierung

Erfassung morphotaktischer Regularitäten: Aufdröseln des Lexikons in *Teillexika*. Jedes Teillexikon kann als *Fortsetzungsklasse eines (Teillexikon-)Eintrags* gewählt werden.

Dekomposition der Verknüpfung der Ebenen (Zwei-Ebenen-Regeln)
Erfassung der *(ortho-)graphischen und phonologischen Abweichungen* der korrespondierenden Zeichen beider Ebenen. Jede Regel drückt eine global gültige Einschränkung zwischen Zeichenpaaren aus.

8.1.2 Morphotaktik

Beispiel 8.1.2 (Morphotaktik des Englischen [**?**]).

- piti-less-ness vs. *piti-ness-less
- un-guard-ed-ly vs. *un-elephant-ed-ly

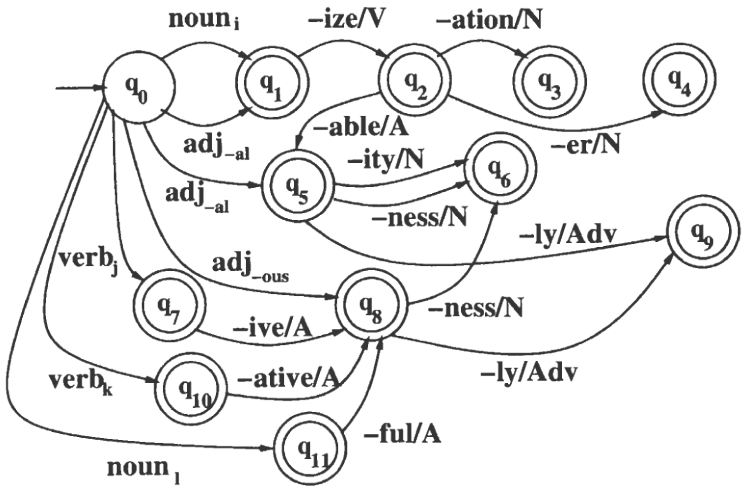


Abbildung 8.3: Morphotaktik des Englischen aus [**?**, 70]

Definition 8.1.3 (Morphotaktik, engl. *morphotactics*)**.** Die *Morphotaktik* einer Sprache bezeichnet die Beschreibung der Kombinierbarkeit ihrer Morpheme (etwa Derivations- und Flexionsmorpheme, Komposition).

8.1.3 Alternation

Orthographische und phonologische Alternanz

Beispiel 8.1.4 (Alternationen des Englischen).

- "pity" wird zu "piti", falls "less" folgt.
- "fly" wird zu "flie", falls "s" folgt.
- "die" wird zu "dy" und "swim" zu "swimm", falls "ing" folgt.

Definition 8.1.5 (engl.*orthographical and phonological alternation*)**.** Die *orthographischen und/oder phonologischen Abweichungen* einer Sprache bezeichnet eine Familie von Phänomenen, bei denen typischerweise an *Morphemgrenzen* kleine Modifikationen stattfinden.

Abgrenzung zu Suppletiv-Formen
Ersatzformenbildung wie in "gut", "besser" oder "viel", "mehr" zählen nicht als Alternanz.

8.2 lexc: Ein Kompiler für Lexika

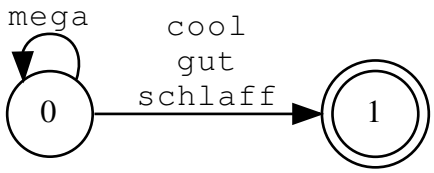
8.2.1 Formalismus

Beispiel: Unbeschränkte Intensivierung aka. Jugendsprache

Adjektiv-Intensivierung mit "mega"

- Adjektive wie "cool", "gut", "schlaff"
- Unbeschränkte Intensivierung durch beliebig wiederholtes "mega"
- "cool","megacool","megamegacool"

Zustandsdiagramm



Ein Formalismus für die lexikalische Ebene

`lexc`-Sprache
Die `lexc`-Sprache ist eine auf lexikographische Bedürfnisse zugeschnittene Notation für endliche Automaten und Transduktoren, welche konkatenative Morphologie inklusive Rechtsrekursion optimal unterstützt.

- Der Kompiler kann *grosse* Lexika effizient verarbeiten.
- Das Konzept der *Teillexika* fördert die klassenbezogene Strukturierung.

- Das Konzept der *Fortsetzungsklassen* auf der Ebene der Lexikoneinträge erlaubt flexible morphotaktische Verknüpfungen.
- Lexikoneinträge können *leer*, d.h. ϵ sein.
- Die Fortsetzungsklasse eines Eintrags kann sich *rekursiv* auf das eigene Teillexikon beziehen.
- Steht in der Tradition der Zwei-Ebenen-Systeme (TwoL von Koskenniemi, PC-KIMMO von [Antworth 1990]).

Aufbau einer lexc-Datei

Beispiel 8.2.1 (Lexikon mit leeren und rekursiven Fortsetzungsklassen ►►►).

```
LEXICON Root    ! Kein ';' hier!
        MEGA ;

LEXICON MEGA
mega    MEGA ; ! Rechtsrekursion
        ADJ  ; ! Leerer Eintrag

LEXICON ADJ
        cool # ;  ! oder
        gut  # ;  ! oder
        schlaff #;
```

Benannte Teillexika
`LEXICON` *`Name`*
Enthalten mindestens 1 Eintrag.
`Root` ist das Startlexikon, `#` das vordefinierte Endlexikon.

Lexikoneinträge
`Entry Continuation` `;`
Entry kann leer sein,
Continuation nicht.

Deklarierung der benutzten Mehrzeichensymbole
Entries müssen nicht in geschweiften Klammern geschrieben werden `Multichar_Symbols` *`MC1 ... MCn`*

Visualisierungen

Zustandsdiagramm der Spezifikation des Mega-Lexikons

Zustandsdiagramm des kompilierten Mega-Lexikons

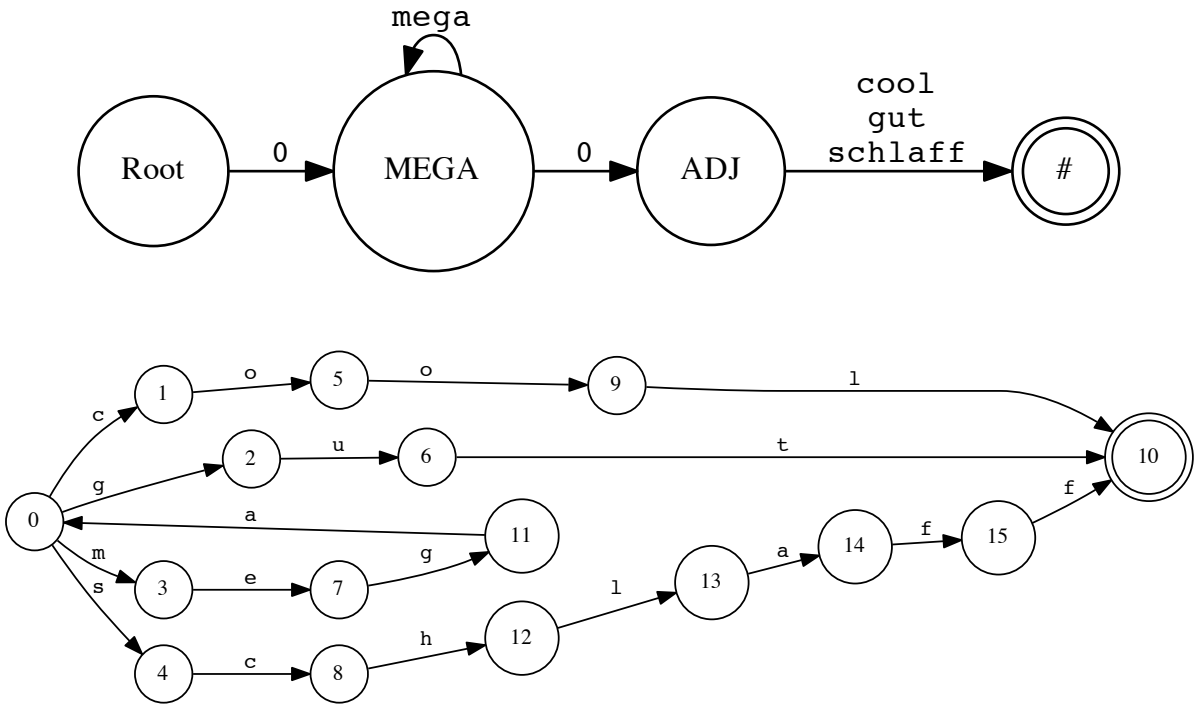


Abbildung 8.4: Zustandsdiagramm eines lexc-Lexikons

Ausschnitt aus GERTWOL ►►►

```
Multichar_Symbols +S +FEM
 +SG +PL +NOM +AKK +DAT +GEN @U

LEXICON Root
* SUBST ;

LEXICON SUBST
mau_s S7+/f;

LEXICON S7+/f
        S7+/f/end;

LEXICON S7+/f/end
        Sg3/f/end;
        Pl1+/f/end;

LEXICON Sg3/f/end
+S+FEM+SG+NOM:0 #;
+S+FEM+SG+AKK:0 #;
+S+FEM+SG+DAT:0 #;
+S+FEM+SG+GEN:0 #;

LEXICON Pl1+/f/end
```

```
+S+FEM+PL+NOM:e@U #;
+S+FEM+PL+AKK:e@U #;
+S+FEM+PL+DAT:en@U #;
+S+FEM+PL+GEN:e@U #;
! @U triggert Umlaut
```

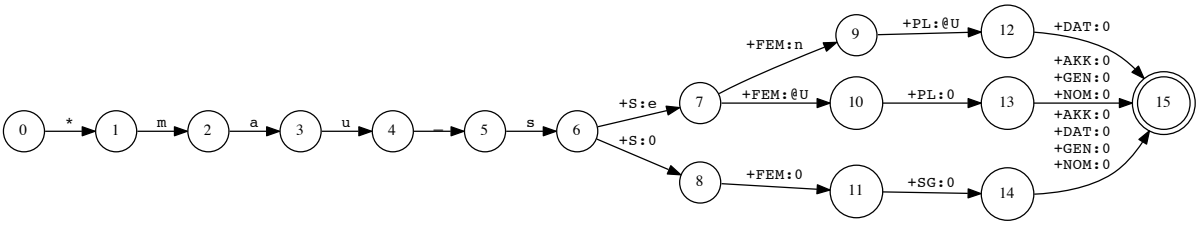


Abbildung 8.5: Zustandsdiagramm des GERTWOL-Ausschnitts

8.2.2 Benutzerschnittstelle

Kompilieren von lexc-Dateien

Mit dem `lexc`-Werkzeug

```
$ lexc
lexc>  compile-source lexicon.lexc
Opening 'lexicon.lexc'...
Root...1, SUBST...1, S7+/f...1, S7+/f/end...2, Sg3/f/end...4, Pl1+/f/end...4
Building lexicon...Minimizing...Done!
SOURCE: 2.7 Kb. 16 states, 22 arcs, 8 paths.
```

Mit dem `xfst/foma`-Werkzeug

```
$ xfst
xfst[0]: read lexc lexicon.lexc
Opening input file 'lexicon.lexc'
Root...1, SUBST...1, S7+/f...1, S7+/f/end...2, Sg3/f/end...4, Pl1+/f/end...4
Building lexicon...Minimizing...Done!
2.7 Kb. 16 states, 22 arcs, 8 paths.
```

Lexikalische Transduktoren mit lexc

Transduktoren

Lexikoneinträge der Form *`Wort1`*`:`*`Wort2`* erzeugen *Transduktoren.*

Anwendungen

- Dies erlaubt eine einfache Spezifikation für Suppletiv-Formen oder stark abweichende Stämme ("go" zu "went").
- In der Zwei-Ebenen-Morphologie wie in GERTWOL heisst die Oberseite dieser Transduktoren Grundform (*base*), die Unterseite heisst lexikalische Ebene (*lexical*). Die Zwei-Ebenen-Regeln verbinden dann die lexikalische Ebene mit der Wortform. Insgesamt ist es eine Drei-Ebenen-Morphologie.

Drei-Ebenen-Morphologie in GERTWOL

Base	Lexical	From lexicon	Surface
*	*	Root	0
m	m	SUBST	M
a	a	SUBST	ä
u	u	SUBST	u
-	-	SUBST	0
s	s	SUBST	s
+S	e	Pl1+/f/end	e
+FEM	@U	Pl1+/f/end	0
+PL	0	Pl1+/f/end	
+NOM	0	Pl1+/f/end	

Abbildung 8.6: 3 Ebenen in GERTWOL [Koskeniemmi und Haapalainen 1996, 123]

8.3 twolc: Ein Formalismus für Alternationen

8.3.1 Formalismus

Ein Formalismus für phonologische Alternationen

`twolc`-Sprache
Die `twolc`-Sprache ist eine auf lexikographische Bedürfnisse zugeschnittene Regel-Notation, um die Bedingungen (ortho-)graphischer und phonologischer Alternationen zu beschreiben.

- Die Zwei-Ebenen-Regeln spezifizieren *Beschränkungen* (*constraints*) über das Vorkommen von *Symbolpaaren*.
- Der `twolc`-Kompiler kann die Regeln automatisch in ET übersetzen.
- Die Regeln müssen immer gleichzeitig *parallel* jedes Symbolpaar eines Zeichenkettenpaars erlauben – ϵ wird dabei wie ein normales Symbol der Länge 1 betrachtet.
- Wegen der Einschränkung auf Vorkommen von ϵ in Symbolpaaren lassen sich die Transduktoren der verschiedenen Regeln als Spezialfall zu einem grossen Transduktor schneiden.

Phänomene mit 2-Ebenen-Regeln in GERTWOL

- *Umlautung* bei Substantiven
- Die *e-Erweiterung* im IND PRÄS SG2/3 und PL2, IMP PRÄS PL2, IND PRÄT und PART PERF bei schwachen Verben, deren Stamm entweder auf -d oder -t endet oder auf -m oder -n ausgeht, vor dem ein anderer Konsonent (kein -l oder -r) steht: reden → redete (nicht: redte); atmen → atmete (nicht: atmte)

- Der *n-Schwund* im PL DAT bei auf -n ausgehenden Substantiven: Laden → Läden (nicht: Lädenn)
- Die (alte) *Rechtschreibung* des ss/ß: Kuß → Küsse; vermissen → vermißt
- Die alte *Rechtschreibung* an der Nahtstelle in Komposita (drei Konsonanten): Schnitt + Tulpe → Schnittulpe

Ausschnitt aus GERTWOL ►►►

```
Alphabet a b c d e f g h i j k l m n o p
        q r s t u v w x y z ä ü @U:0 #:0 %_:0 ;

Rules
"Uml a~ä"
 a:ä <=> _  [ \ [#: | a: | o: | u: ]]* @U: ;
        _ u: [ \ [#: | a: | o: | u: ]]* @U: ;

"Uml u~ü"
 u:ü <=>  \a: _  [#: | a: | o: | u: ]* @U: ;
```

Notationen der Form `"c:"` (oder `":c"`) bezeichnen nicht die Identitätsrelation, sondern alle möglichen Symbolpaare aus dem Alphabet (*feasible pairs*), welche auf der einen Seite das Symbol `c` aufweisen.

Aufbau einer twolc-Datei

Alphabet (*feasible pairs*)
Nach dem Schlüsselwort `Alphabet` werden alle möglichen Symbolpaare des Regeltransduktors aufgelistet.

Regeln
Nach dem Schlüsselwort `Rules` folgen die Regeln, denen jeweils ein treffender Name in Anführungszeichen vorangeht.

Reguläre Konstrukte
In den Regeln sind die meisten regulären Operatoren für reguläre Sprachen (nicht Relationen!) zugelassen. Sie dürfen aber *nie* nur auf eine Seite eines Symbolpaars angewendet werden: Verboten: `a:[b+]`.
Im Gegensatz zu den Ersetzungsregeln von `xfst` dürfen alle Bestandteile sich auf die obere und die untere Sprache beziehen (deshalb: Zwei-Ebenen-Regeln).

Semantik der Regel-Operatoren

8.3.2 Benutzerschnittstelle

Kompilieren von twolc-Dateien

Einlesen und kompilieren

	Positive Reading	*Negative Reading*
a:b <=> l _ r ;	1. If the symbol pair a:b appears, it must be in the context l _ r. 2. If lexical a appears in the context l _ r, then it must be be realized on the surface as b.	1. If the symbol pair a:b appears outside the context l _ r, FAIL. 2. If lexical a appears in the context l _ r and is realized as anything other than b, FAIL.
a:b => l _ r ;	If the symbol pair a:b appears, it must be in the context l _ r.	If the symbol pair a:b appears outside the context l _ r, FAIL.
a:b <= l _ r ;	If lexical a appears in the context l _ r, it must be realized on the surface as b.	If lexical a appears in the context l _ r and is realized as anything other than b, FAIL.
a:b /<= l _ r ;	Lexical a is never realized as b in the context l _ r .	If lexical a is realized as b in the context l _ r, FAIL.

Abbildung 8.7: Semantik der 2-Ebenen-Operatoren

```
$ twolc
twolc>  read-grammar rules.twol
reading from "rules.twol"...
Alphabet... Rules...
  "Uml a~ä" "Uml u~ü"
twolc>  compile
Compiling "rules.twol"
Expanding... Analyzing...
Compiling rule components:
  "Uml a~ä" "Uml u~ü"
Compiling rules:
  "Uml a~ä"
    a:ä <=> 2 1
  "Uml u~ü"
    u:ü <=>
Done.
```

Inspizieren und exportieren

```
twolc>  show-rules
"Uml a~ä"
   ? a o u a:ä @U:0
```

```
1: 1 2 1 1  3    1
2: 5 2 1 5  3
3. 4     4       1
4. 4             1
5: 5 2 1 1  3
Equivalence classes:
((? b c d e f g h i j k l m n p
q r s t v w x y z ä ü _:0) (a)
(o #) (u u:ü) (a:ä) (@U:0))
...
twolc>  save-binary rules.fst
Writing "rules.fst"
```

Zustandsdiagramm für Regel "Uml a~ä"

```
"Uml a~ä"
 a:ä <=> _  [ \ [#: | a: | o: | u: ]]* @U: ;
        _ u: [ \ [#: | a: | o: | u: ]]* @U: ;
```

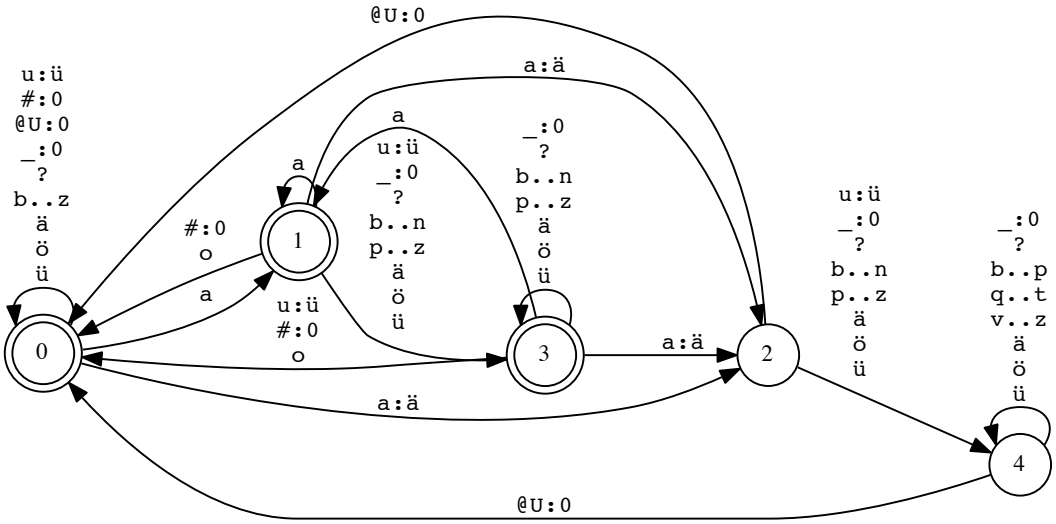


Abbildung 8.8: Zustandsdiagramm für Zwei-Ebenen-Regel

Zusammenfügen von Lexikon und Regeln

Einlesen

```
$ lexc
lexc> compile-source lexicon.lexc
...
lexc>  random-surf
NOTE:  Using SOURCE.
*mau_se@U
*mau_se@U
```

```
...
lexc>  random-lex
NOTE:  Using SOURCE.
*mau_s+S+FEM+SG+NOM
*mau_s+S+FEM+SG+NOM
...
lexc>  read-rules rules.fst
Opening 'rules.fst'...
2 nets read.
```

Verknüpfen und exportieren

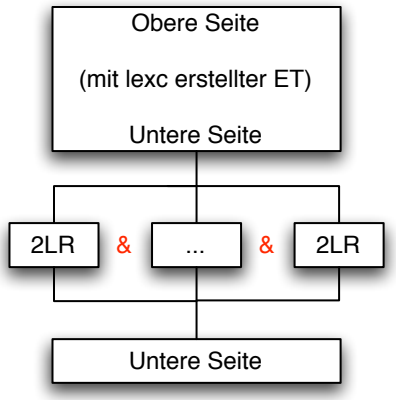
```
lexc>  compose-result
No epenthesis.
Initial and final word boundaries
added. ..Done.
1.9 Kb. 27 states, 35 arcs, 8 paths.
Minimizing...Done.
1.7 Kb. 20 states, 26 arcs, 8 paths.
lexc>  random-surf
Use (s)ource or (r)esult? [r]: r
NOTE:  Using RESULT.
*maus
*mäusen
...
lexc>  save-result lextrans.fst
Opening 'lextrans.fst'...
Done.
```

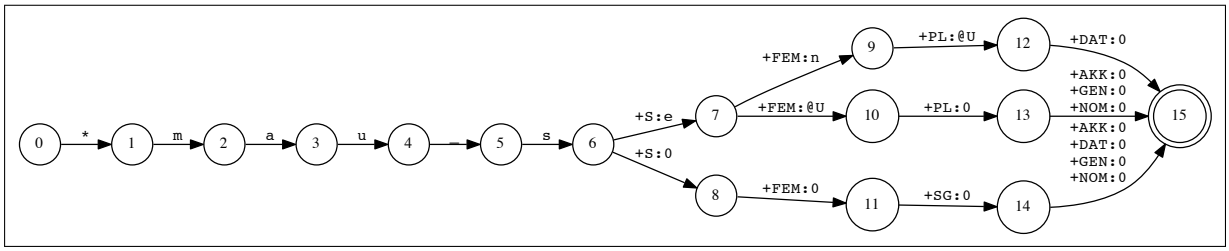
8.4 Zwei-Ebenen-Transduktoren

Eine graphische Übersicht zum vorangegangenen Kode

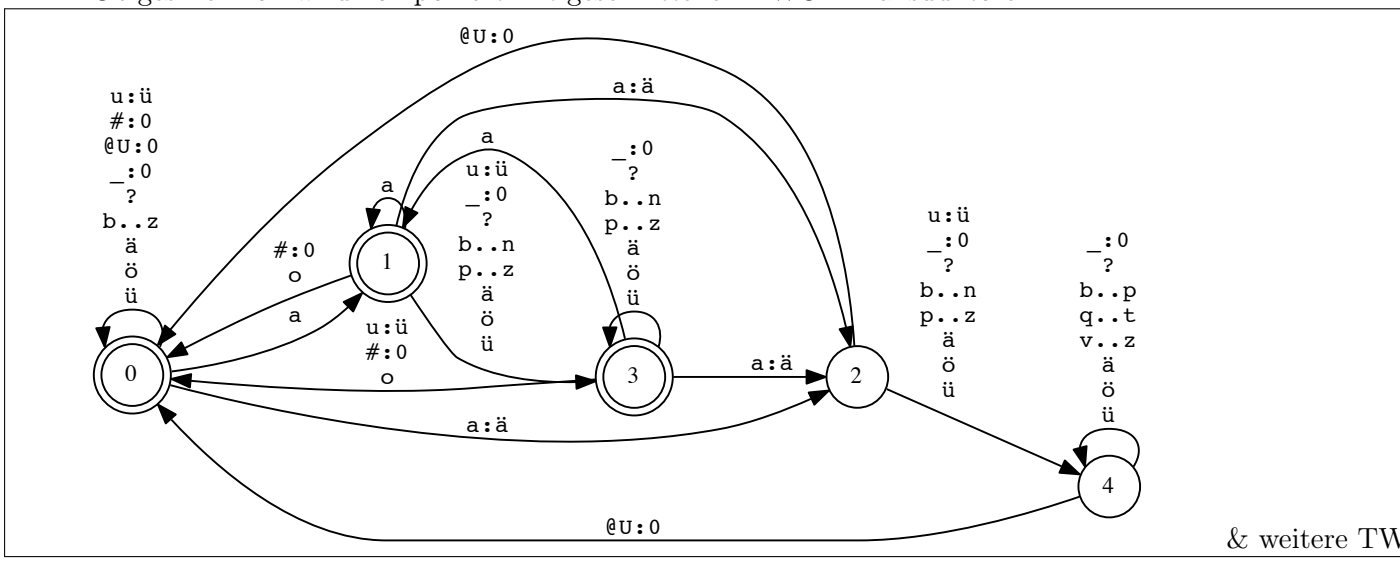
8.4.1 Seriell oder parallel?

Äquivalenz von paralleler und serieller Dekomposition





Obiges Lexikon wird komponiert mit geschnittenen TWOL-Transduktoren



& weitere TW

und ergibt

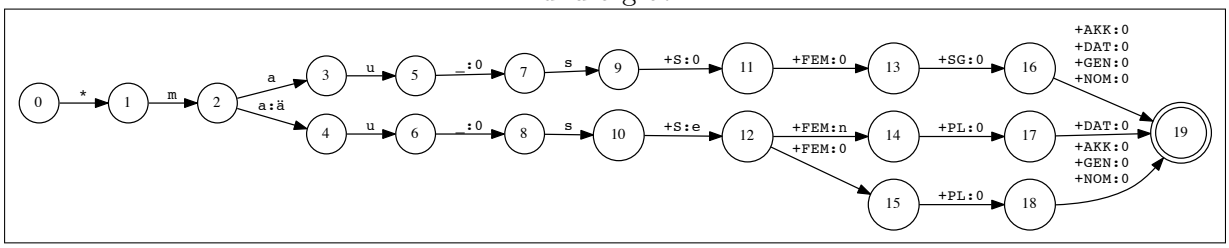
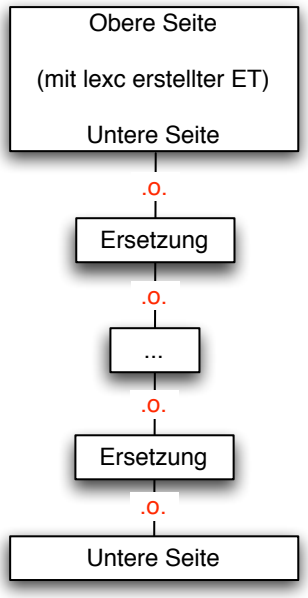


Abbildung 8.9: Diagramm des komponierten Transduktoren

Schneide die 2-Ebenen-Regeln (2LR)!



Komponiere die Ersetzungsregeln!

Zwei-Ebenen-Regeln oder Ersetzungsregeln?
Ob die reguläre Relation zwischen lexikalischer Ebene und Wortform mit Zwei-Ebenen-Regeln oder komponierten Ersetzungsregeln dekomponiert wird, ist mit den mächtigen Kompilern und leistungsfähigen Computern eine Stilfrage geworden.

Vorteile und Nachteile

- *Sequentielles Denken* liegt den meisten Leuten näher als paralleles.
- Ersetzungsregeln sind für sich genommen *eingeschränkter*, weil sie sich nicht direkt auf beide Seiten beziehen können.
- Zwei-Ebenen-Regeln können in schwierig aufzulösende *Konflikte* kommen.

8.5 Vertiefung

- Das Handbuch [Beesley und Karttunen 2003c] zu `twolc` ist ein Kapitel, das nicht in [Beesley und Karttunen 2003b] aufgenommen wurde. Wer sich ernsthaft mit Zwei-Ebenen-Morphologie (2LM) beschäftigen will, arbeitet es durch. Gleiches gilt für die `lexc`-Sprache, welche wegen der ebenfalls leicht abweichenden Syntax zu `xfst` einige Fallen bereitstellt.
- Wer die Implementation eines umfassenden und hochwertigen 2LM-Systems verstehen will, macht sich in der informativen `http://files.ifi.uzh.ch/CL/volk/LexMorphVorl/Lexikon04.Gertwol.html` oder bei [Koskeniemmi und Haapalainen 1996] schlau. Allerdings verwenden sie nicht `twolc`, sondern das von Koskenniemi entwickelt TwoL-Werkzeug, welches eine leicht andere Syntax hat. Die Übertragung auf `lexc` und `twolc` ist leicht.
- [Carstensen et al. 2004, 173ff.] präsentiert einige der Ideen kompakt, aber ohne Implementation.

Kapitel 9

Nicht-konkatenative Morphologie

Lernziele

- Kenntnis über den Einsatz von eleganten Techniken zur Modellierung nicht-konkatenativer Morphologie
- Kenntnis über die Verwendung von *flag diacritics*, d.h. den Mechanismus zur Manipulation und Unifikation von Merkmalwertpaaren
- Erkenntnis über den Unterschied von technisch machbarer und theoretisch befriedigend generalisierter Modellierung von schwierigen morphologischen Erscheinungen

9.1 Nicht-konkatenative Morphologie (NKM)

Konkatenative vs. nicht-konkatenative Morphologie

Theorie
Rein konkatenative EA-Morphologie verwendet nur Konkatenation und Vereinigung, wie sie in `lexc` zur Verfügung gestellt wird, um morphologische Einheiten zu kombinieren.

Praxis
Natürliche Sprachen (auch agglutinierende) funktionieren aber nie so: *Assimilation* (Lautanpassung), *Epenthese* (Lauteinschaltung) und *Elision* (Lautausfall) ergeben sich aus dem Zusammenspiel von Morphemen.

Extrem konkatenative Morphologie

Beispiel 9.1.1 (Polysynthetische Sprache: Inuktitut).

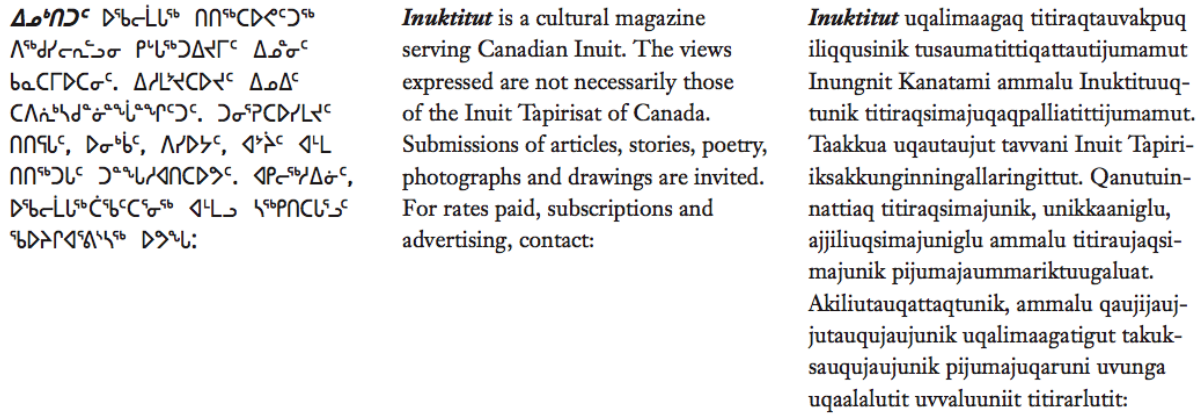
Oben	Paris+mut+nngau+juma+niraq+lauq+sima+nngit+junga
Glosse	Paris Terminalis-Kasus Weg-nach wollen sagen-dass Vergangenheit Perfektiv Zustand Negativ 1sg-intransitiv
Unten	Parimunngaujumaniralauqsimanngittunga
Bed.	Ich sagte niemals, dass ich nach Paris gehen will.

Quelle: [Beesley und Karttunen 2003b, 376].

Wo findet sich hier Assimilation, Epenthese oder Elision?

Morphologie in einer polysynthetischen Sprache
Eine gute deutschsprachige Einführung in Inuktitut gibt [Nowak 2002]. Eine Affixliste dazu findet sich in [Ehrhardt 2003], wo man die behaupteten Fakten zu verifizieren versuchen kann.

Beispiel für normales modernes geschriebenes Inuktitut



ᐃᓄᒃᑎᑦ ᐅᖃᓕᒫᖅ ᑎᑎᖅᑕᐅᓯᔪᓐᓇᖅ ᐱᖅᓯᓂᓪᓗᓂᓂᓗ ᑭᒫᖅᑐᐃᔨᒥᑦ ᐃᓄᒋᓂᑦ ᖃᓄᐃᒥᑕᐅᓂᑦ. ᐃᓱᒪᔨᑕᐅᔨᑦ ᐃᓄᐃᑦ ᑕᐱᓵᓴᓐᓂᖅᓯᓂᑦ. ᑐᓂᖏᑕᐅᓱᒪᔨᑦ ᑎᑎᕐᒪᑦ, ᐅᓂᒃᑲᑦ, ᐱᓱᐅᔪᑦ, ᐊᑦᔨᑦ ᐊᒻᒪ ᑎᑎᖅᑐᒪᑦ ᑐᖓᓕᓱᐊᑎᑕᐅᔪᑦ. ᐊᑭᓚᖅᓱᐃᓂᑦ, ᐅᖃᓕᒫᖅᑕᖏᖅ ᐊᒻᒪᓗ ᓴᖅᑭᑎᑕᒪᓕᔪᑦ ᖃᓄᖅ ᐅᖃᓕᒫᖅᑐᑦ ᖃᓄᐃᓕᓴᐅᔪᖅ ᐅᖓᓪᓗ:

Inuktitut is a cultural magazine serving Canadian Inuit. The views expressed are not necessarily those of the Inuit Tapirisat of Canada. Submissions of articles, stories, poetry, photographs and drawings are invited. For rates paid, subscriptions and advertising, contact:

Inuktitut uqalimaagaq titiraqtauvakpuq iliqqusinik tusaumatittiqattautijumamut Inungnit Kanatami ammalu Inuktituuqtunik titiraqsimajuqaqpalliatittijumamut. Taakkua uqautaujut tavvani Inuit Tapiriiksakkunginningallaringittut. Qanutuinnattiaq titiraqsimajunik, unikkaaniglu, ajjiliuqsimajuniglu ammalu titiraujaqsimajunik pijumajaummariktuugaluat. Akiliutauqattaqtunik, ammalu qaujijaujjutauqujaujunik uqalimaagatigut takuksauqujaujunik pijumajuqaruni uvunga uqaalalutit uvvaluuniit titirarlutit:

Quelle: `http://www.itk.ca/inuk-mag/magazines/InukMagazine85.pdf`

Abbildung 9.1: Beispiel für normales modernes geschriebenes Inuktitut

Nicht-konkatenative Morphologie
Typischerweise versteht man darunter Phänomene bzw. eine theoretische Beschreibung davon, welche mit Affigierung von Morphen allein keine genügende Abstraktion liefert.

- *Infixe* (Modifikationen innerhalb von Morphen): *ich singe, sang* (Ablaut im Germanischen)
- *Zirkumfixe* (nicht-lokale Abhängigkeit von zwei Morphen): *ge+spann+t*,
- *Nicht-lokale Abhängigkeiten/Beschränkungen*: **unarbeitslos*
- *Reduplikation*: Wiederholung von Silben oder ganzen Wörtern: *pend-o, pe-pend-i* (in Latein für ich hänge, hing), *rumah-rumah* (im Malay, Plural von Haus)
- *Root-Pattern-Ansatz* (*Transfigierung (Interdigitation)* von Morphemen) für semitische Sprachen: katab = k-t-b + -a-a- (Arabischer Stamm)

Konkatenative vs. nicht-konkatenative Sprachen

Ist alles konkatenativ?
Wegen der phonetischen bzw. graphematischen Serialisierung von Sprache kann grundsätzlich alles konkatenativ beschrieben werden.

Beschreiben = Strukturieren und Generalisieren
Die Morphologie will aber Beziehungen zwischen morphologischen Einheiten identifizieren und systematisch verallgemeinernd beschreiben.
Worin gleichen/unterscheiden sich flektierte, derivierte Formen? Welche semantischen Bezüge lassen sich (nicht) ausdrücken?

Elegante Theorie und Implementation
Gewisse Phänomene lassen sich nun mal viel eleganter und allgemeiner mittels nicht-konkatenativer Konzepte beschreiben.

9.1.1 Einfache Reduplikation

Reduplikation mit Ersetzungsregeln

Beispiel 9.1.2 (Reduplikation eines Morphems fester Länge in Tagalog nach [Antworth 1990]).

Wurzel	CV+Wurzel	Bedeutung
pili	pipili	wählen
kuha	kukuha	nehmen

Zum Bedeutungsaspekt der reduplizierten Formen siehe etwa `http://www.seasite.niu.edu/TAGALog/tagalog_verbs.htm`.

Vereinfachtes Tagalog-Beispiel ►►►

```
define AbsVerbs [
   (R E "+")              ! Optionale abstrakte Reduplikation
   [{pili} | {kuha}]      ! Verbwurzeln
];

define Cs [p|k];          ! Konsonanten

define RRule1 [R -> p || _ E "+" p];
define RRule2 [R -> k || _ E "+" k];

define ERule1 [E -> i || _ "+" Cs i];
define ERule2 [E -> u || _ "+" Cs u];

define Cleanup ["+" -> 0];

define Verbs [ AbsVerbs
 .o. RRule1 .o. RRule2    ! Zuerst R konkretisieren
 .o. ERule1 .o. ERule2    ! Dann E
 .o. Cleanup
];
```

9.2 Flag-Diacritics

9.2.1 Unifikation

Flag-Diacritics (diakritische Merkmale)

Definition 9.2.1. *Flag-Diacritics* sind Mehrzeichen-Symbole, welche zur Laufzeit an den mit ihnen beschrifteten Kantenübergängen Instantiierungen und Zuweisung von atomaren Werten an eine Variable auslösen. Ansonsten funktionieren sie aber wie ein ϵ-*Übergang*.

Unifikationstest an einem Kantenübergang

`"@U.`*FEATURE*`.`*VALUE*`@"`

Falls das Merkmal ***FEATURE*** uninstantiiert ist, unifiziere es mit dem Wert ***VALUE*** und erlaube den Kantenübergang. Falls das Merkmal ***FEATURE*** instantiiert ist, erlaube den Kantenübergang nur, wenn es den Wert ***VALUE*** hat.

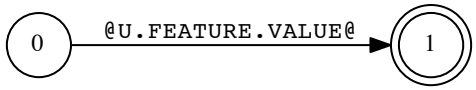


Abbildung 9.2: EA mit Unifikationstest-Übergang

Einfaches Beispiel ►►►

Beispiel 9.2.2 (Flag-Diacritic für Unifikationstest).

```
read regex [
  [ a "@U.merkmal.a@" | b "@U.merkmal.b@" ]
  c
  [ "@U.merkmal.a@" a | "@U.merkmal.b@" b ]
];
```

Fragen

- Welche Wörter akzeptiert dieser reguläre Ausdruck?
- Wieviele Pfade hat das Übergangsdiagramm?

Linguistisches Beispiel in lexc ►►►

```
Multichar_Symbols @U.ART.PRESENT@ @U.ART.ABSENT@ uN aN iN

LEXICON Root
    Article ;

LEXICON Article                          ! "al" ist optionaler definiter Artikel,
al @U.ART.PRESENT@ Stems ;         ! dessen Präsenz Indefinit-Endungen verbietet
                   Stems ;                                ! Kein definiter Artikel

LEXICON Stems
kitaab Case ;                                             ! Beispieleintrag "Buch"

LEXICON Case
u #;                                                        ! Definiter Nominativ
a #;                                                        ! Definiter Akkusativ
i #;                                                          ! Definiter Genitiv
@U.ART.ABSENT@ IndefCase ;                      ! Falls definiter Artikel fehlt,
                                          ! ist Suffix für Indefinit-Endung möglich
LEXICON IndefCase
uN # ;                                                     ! Indefiniter Nominativ
aN # ;                                                     ! Indefiniter Akkusativ
iN # ;                                                       ! Indefiniter Genitiv
```

9.2.2 Weitere Operatoren

Weitere Tests und Operatoren

- `"@P.MERKMAL.WERT@"`: Setze (*positive set*) MERKMAL auf WERT! Gelingt immer.
- `"@C.MERKMAL@"`: Deinstantiiere (*clear*) MERKMAL! Gelingt immer.
- `"@D.MERKMAL@"`: Blockiere (*disallow*), falls MERKMAL instantiiert ist!

Beispiel 9.2.3 (Technik: Forward-Looking Requirements mit `@P`, `@C` und `@D`). Aktiviere Forderung für die Präsenz eines nachfolgenden Elements.
Lösche Forderung, wenn sie erfüllt ist. Ein Schlusstest verbietet alle Pfade, bei denen die Forderung noch besteht.

```
LEXICON Foo
foo@P.RequireBar.ON@ CC1;
...                                          ! Forderung aktiviert
LEXICON Bar
bar@C.RequireBar@ CC2;
...                                          ! Forderung erfüllt
LEXICON CEnd
@D.RequireBar@ # ;
! Blockiere Pfade mit aktiver Forderung
```

Vorteil von Flag-Diacritics

- Nicht-lokale Abhängigkeiten lassen sich *elegant* ausdrücken, indem sie in Merkmalswertbelegungen kodiert werden.
- Tests auf Flag-Diacritics sind sehr *effizient* implementiert – im Unterschied zur vollen Unifikation wie sie etwa im PC-KIMMO-System unterstützt wird.
- Flag-Diacritics lassen sich immer aus einem EA/ET *eliminieren*:
  `xfst[1]: eliminate flag ART`  (momentan buggy in xfst)
- Flag-Diacritics in `lexc` erlauben *Klassifikationen*, welche *orthogonal* zu den Fortsetzungsklassen verlaufen (semantische Kriterien, Derivationbeschränkungen)
- *Effizienz*: Die Grösse von Transduktoren kann mit geeigneten diakritischen Merkmalen entscheidend beeinflusst werden. (Morphologie für Ungarisch: 5 Merkmale reduzieren 38MB Transduktor auf 5 MB [Beesley 1998])

9.2.3 Elimination diakritischer Merkmale

Manuelle Eliminierung von diakritischen Merkmalen ►►►
Diakritische Merkmal definieren *Restriktionen* über einer Sprache:

```
reg Restr ~ [           ! Es darf nicht sein, dass
[$ "@U.ART.PRESENT@"]   ! ein ABSENT nach einem PRESENT steht.
  ~ $ ["@P.ART.ABSENT@" ! Solange nicht ABSENT dazwischen positiv gesetzt
       | "@C.ART@"      !  oder das Merkmal ART neutralisiert ist.
     ]
[$ "@U.ART.ABSENT@" ]
];
```

Vorgehen zur Eliminierung
Die Restriktion muss auf den Transduktor komponiert werden. Danach lassen sich alle diakritischen Merkmale mit Bezug auf ART durch ϵ ersetzen.

9.3 compile-replace-Technik

9.3.1 Sondernotation und der Befehl

Compile-Replace-Befehl
Reguläre Ausdrücke `RA` zwischen `"^[" RA "^]"` auf einer Seite eines Transduktors können sekundär kompiliert und durch ihr Kompilat im Transduktor ersetzt werden.

Beispiel 9.3.1 (Stackbefehl `compile-replace lower` in xfst).

```
reg  u:["^[" l "+" "^]"];


compile-replace lower
```

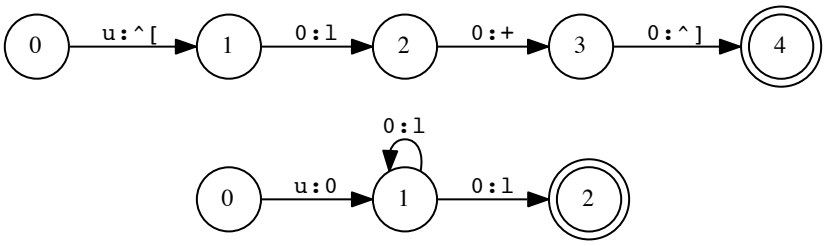


Abbildung 9.3: Primär und sekundär kompilierter regulärer Ausdruck

Fragen
Wieviele Wörter erkennen die beiden Transduktoren?

9.3.2 Reduplikation nochmals

Compile-Replace für unbegrenzte Reduplikation

Beispiel 9.3.2 (Reduplikation zur Pluralbildung im Malayischen).

Stamm	Reduplikation	Bedeutung
buku	buku-buku	Buch
pelabuhan	pelabuhan-pelabuhan	Hafen

Die Stämme sind unmarkiert bezüglich Numerus. Mit der Reduplikation des ganzen Stammes entsteht eine morphologisch markierte Pluralform.

Abstrakte Pluralbildung für compile-replace-Einsatz ►►►

Idee
Im Plural wird der Stamm 2-fach konkateniert auf der unteren Sprache:

```
{buku}^2
```

Alignierung

```
Upper: 0  0 b u k u +Plural   0
       |  | | | | | |         |
Lower: ^[ { b u k u }^2       ^]
```

```
define NRoots {buku}|{pelabuhan};

define RDPrefix "^[" "{";
define RDSuffix "}^2" "^]";

define Num [
   "+Unmarked":0
 | "+Plural":RDSuffix
];

define AbsNoun [
  ( 0:RDPrefix ) NRoots Num
];
```

Problem

Die untere Sprache erlaubt Wörter, welche keine kompilierbaren RA ergeben. Welche? `buku}^2^]`

Untere Sprache kompilierbar machen und kompilieren

Ungrammatisches herausfiltern

```
define BracketFilter [ ~ [       ! Niemals:
   [ ?* RDPrefix ~$[RDSuffix] ]  ! Ein Prefix ohne nachfolgenden Suffix
 | [ ~$[RDPrefix] RDSuffix ?* ]  ! oder ein Suffix ohne vorangehendes Prefix
 ]];
```

Alles zusammensetzen

```
read regex AbsNoun .o. BracketFilter;

compile-replace lower

apply up bukubuku
apply up buku
```

Vollreduplikation mit Marking und compile-replace ►►►

```
define NRoots {buku}|{pelabuhan};

define Pluralize [
 NRoots -> "^[" "{" ... "}^2" "^]"
];
```

```
read regex [
  NRoots "+Unmarked":0
 | [ NRoots "+Plural":0 .o. Pluralize ]
];

compile-replace lower
```

9.4 Interdigitation mit `merge` und `list`

9.4.1 Semitische Morphologie

Semitische Interdigitation im 3-Ebenen-Modell ▶▶▶
Die Analyse von McCarthy (1981) setzt für semitische Sprachen eine Abstraktion auf 3 Ebenen an:

- *Wurzel* (*root*): k t b
- *CV-Muster* (*template*): CVCVC
- *Vokalfüllung* (*vocalization*): u i
- *Stamm* (*stem*): kutib

9.4.2 Transfigierung durch Interdigitation

Ein Operator für Interdigitation

Supersymbole
Der xfst-Befehl `list` *`Symbol Zeichenliste`* definiert alle möglichen Zeichenbelegungen für *`Symbol`*.

```
xfst[0]: list C b t y k l m n f w r z d s ;
xfst[0]: list V a i u ;
```

Die beiden zweistelligen merge-Operatoren
`RA1` `.<m.` *`RA2`* ersetzt in einer Sprache *`RA1`* mit Supersymbolen diese Symbole der Reihe nach durch die Zeichen aus *`RA2`*.

```
xfst[0]: read regex {CVCVC} .<m. {ui};
xfst[1]: print words
CuCiC
```

Der Operator `.m>.` macht dasselbe mit vertauschten Operanden.

```
# Nach Beesley & Karttunen 2003: 404ff.

list C b t y k l m n f w r z d s ;
list V a i u ;

define Root {ktb}|{drs};

define Template "+FormI":{CVCVC} | "+FormIII":{CVVCVC};
```

```
define Vocalization "+Pass":"[u*i]" | "+Act":"[a+]";

define PerfSuffix ["+3P" "+Masc" "+Sg"]:a | ["+3P" "+Fem" "+Sg"]:{at};

read regex "[":"^["  ! compile-replace
   0:"{" Root 0:"}"
   0:".m>."          ! merge Root ins Template
   0:"{" Template 0:"}"
   0:".<m."          ! merge Vokalization ins Template
   Vocalization
   "]":"^]"
   PerfSuffix ;

compile-replace lower

apply up kataba
```

9.5 Vertiefung

Kapitel 7 und 8 von [Beesley und Karttunen 2003b] sind Pflichtlektüre für alle, welche sich für fortgeschrittene Techniken der Xerox-Tools interessieren. Dazu muss man aber die Basis-Techniken gut verstehen!
Operatoren wie `merge` sind Werkzeuge, welche spezifisch für die Problematik von bestimmten Sprachfamilien entwickelt wurden.
Der Einsatz von diakritischen Merkmalen sollte besonders gut geplant und motiviert sein.
Die neueste Version von xfst unterstützt noch weitere Funktionalität in diakritischen Merkmalen.

Kapitel 10

Was ist Morphologie? II

Herzlichen Dank an Manfred Klenner für Quelltexte.

Lernziele

- Kenntnis der strukturalistischen Begriffe Morph, Allomorph und Morphem und ihrer Tücken in der Verwendung
- Kenntnis des Unterschieds von Silbe und Morph
- Kenntnis der morphologischen Sprachtypologie: isolierend, flektierend, agglutinierend, polysynthetisch
- Kenntnis der morphologischen Prozesse wie Affigierung, Reduplikation oder Suppletion; sowie ihrer Klassifikationskriterien
- Kenntnis über Themen und Anwendungen der Computermorphologie

10.1 Strukturalistische Morphologie

Analyseebenen für Wörter

Segmentationsebenen für Wörter

- *Buchstaben*: a-b-g-e-s-a-g-t
- *Laute/Phone*: a-p-g-ə-z-aː-k-t
- *Silben*: ab-ge-sagt
- *Morphe*: ab-ge-sag-t

10.1.1 Morph

Der Begriff *Morph*

Definition 10.1.1. Ein *Morph* ist die kleinste segmentierbare Einheit einer Wortform, welche semantische oder grammatische Information trägt. Sie wird als *Lautform* (phonetisch) oder als *Schriftform* (graphematisch) aufgefasst.

Segmentieren und Klassifizieren
Mit der Minimalpaaranalyse identifiziert man die Laute (Phone), welche als Klasse (Phonem) die kleinsten bedeutungs*unterscheidenden* abstrakten Einheiten darstellen.
Diese Verfahren wird auf die Ebene der kleinsten bedeutungs*tragenden* Einheiten übertragen.

Anzahl Morphe in einer Sprache
Eine Sprache umfasst typischerweise einige 10'000 Morphe. Die Anzahl Wörter ist eine Grössenordnung höher.

10.1.2 Allomorph

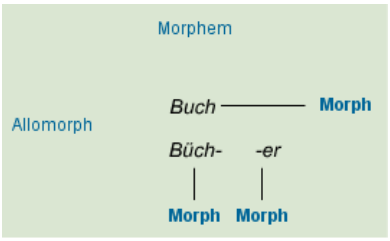
Definition 10.1.2 (auch Morphemvariante)**.** *Allomorphe* sind Morphe, welche sich zwar lautlich oder graphematisch leicht unterscheiden, aber trotzdem die gleiche semantische oder grammatische Funktion wahrnehmen können.

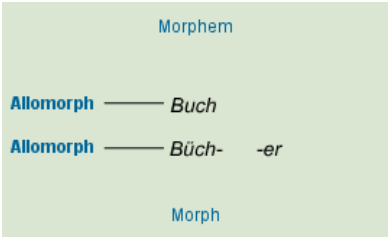
Beispiel 10.1.3 (Semantische Allomorphe).
Die Morphe "buch" und "büch" bedeuten beide Buch.

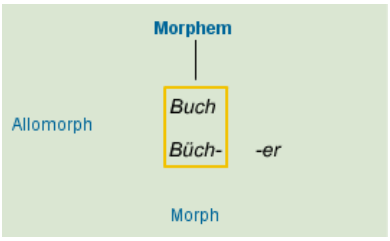
Beispiel 10.1.4 (Grammatische Allomorphe).
Die Morphe "en", "n" oder "e" tragen Pluralinformation im Deutschen.

10.1.3 Morphem

Definition 10.1.5 (klassisch strukturalistische Definition)**.** Ein *Morphem* ist die kleinste bedeutungstragende Einheit des Sprachsystems.







Beispiel 10.1.6 (Morph vs. Allomorph vs. Morphem [Stocker et al. 2004]).

Abstraktionsgrad von Morphem
Ein Morphem ist eine *Menge von Morphen*, welche Allomorphe voneinander sind, d.h. die gleiche semantische oder grammatische Funktion ausüben.

Morphem

Ursprung nach [Glück 2000]
"L. Bloomfield bestimmte das Morphem als »minimale Form«, eine Phonemfolge, die nicht in kleinere Einheiten zerlegbar ist und die eine feste Bedeutung besitzt" (Language 1933, 158-161)

Entstehung des Begriffs Morphem
"Von J. Baudouin de Courtenay um 1880 geprägter Oberbegriff für Beschreibungsbegriffe der herkömmlichen »Formenlehre« wie Vorsilbe, Nachsilbe, Endung oder Stamm (von Wörtern)." [Glück 2000]

Abbildung 10.1: Leonard Bloomfield (1887–1949)

Trennung und Vermischung von Form und Funktion

- Die Wörter auf "-em" (Phonem, Morphem, Lexem) beziehen sich auf die (theorieabhängig postulierten) *Funktionen* im Sprachsystem.
- Das Wort "Morph" (von griechisch $\mu o \rho \phi \eta$ "Form") oder Phon bezieht sich auf die *Ausdrucks- oder Formseite* der Sprachen.
- Oft verwendet man (verkürzend) den Begriff Morphem auch für die Ausdrucksseite, d.h. für die Morphe.
- Oft verwendet man die graphematische Form von Morphen als Symbol für Morpheme: Das Morphem "buch" hat die zwei Allomorphe "Buch" und "Büch".
- Eine sauberere Notationsmöglichkeit wendet [Kunze 2007] an: Morphe werden in spitzen Klammern notiert, Morpheme in geschweiften.
- Das Morphem {buch} hat die Allomorphe <buch> und <büch>.

Morph(em)-Zoo im Überblick (nach [Bussmann 2002])

Bedeutungseigenschaft

- *Lexikalische M.* (Lexeme) bezeichnen Aussersprachliches: {brot}. Untersuchungsgebiet der Semantik und Lexikologie.
- *Grammatische M.* (Flexionsmorph(em)e) drücken innersprachliche Beziehungen aus: {und},{PLURAL}. Untersuchungsgebiet der Morphologie und Syntax.

Distributionseigenschaft (Vorkommen, Selbständigkeit)

- *Gebundene M.* sind Stamm-, Flexions- oder Derivations-Morph(eme), welche nur mit weiteren M. eine Wortform bilden: {zer-}
- *Unikale M.* kommen als gebundene M. nur in einer Kombination vor: {him}
- *Freie M.* bilden für sich Wortformen.

Einheit von Form und Funktion

- *Diskontinuierliche M.* bestehen aus mehreren "getrennten Morphen": Das Morphem {PARTIZIP} in der Wortform "getrennt" besteht aus "<ge>"+"<t>".
- *Portmanteau-M.* vereinigen die Bedeutung/Funktion mehrerer Morph(em)e in sich: "schrieb" ({PAST},{schreiben}, ...)

Frage
Wo würde man statt von Morphemen besser von Morphen sprechen, wenn die Unterscheidung von Form und Funktion ernst genommen wird?

Lexikalisierung und Demotivierung

Definition 10.1.7 (Lexikalisierung)**.** Mit *Lexikalisierung* drückt man aus, ob ein Wort (oder eine grössere syntaktische Einheit) als ein Zeichen im (mentalen) Lexikon gespeichert ist.

Definition 10.1.8 (Demotivierung, Idiomatisierung)**.** Die *Demotivierung* bezeichnet ein Phänomen des Sprachwandels, dass mehrgliedrig analysierbare Ausdrücke zu einer lexikalischen Einheit werden, deren Gesamtbedeutung nicht mehr aus den Bestandteilen erkennbar (motiviert) ist.

Beispiel 10.1.9 (Demotivierung).

- Kunst aus "können" und t-Suffix: Was bedeutet das t-Morph?
- "erklecklich" aus ?: Eine Analyse als <er><kleck><lich>scheint formal sinnvoll. Warum? – bedeutungsmässig ergibt sich daraus nichts.

(De)Motivierung bei geographischen Bezeichnungen

Beispiel 10.1.10 (Etymologische Kolumne "Besserwisser" vom Tages-Anzeiger vom 26.3.2007 Seite 66).
Was macht den Hügel froh?
Die Rebenstrasse in Leimbach heisst so, weil dort einst Reben wuchsen. Der Tannenweg in Altstetten führt durch den Tannenwald. Und Im Schilf in Fluntern stand vor langer Zeit ein grosses Schilfgebüsch. Falsch. Zürichs Strassennamen sind manchmal doppelbödig. Dieses Schilf meint eigentlich Schülff, und das war der Übername des Landbesitzers im 14. Jahrhundert, Johann Bilgeri.
Im Hummel in Wollishofen will nicht an die vielen Hummeln erinnern, die es dort einmal gab, sondern an den früheren Flurnamen: Humbel, abgeleitet aus Hohenbüel. Büel bedeutet Hügel, Anhöhe, weshalb die Frohbühlstrasse in Seebach auf einen frohgemuten Hügel hinweist und die Frage auf wirft: Was stimmt einen Hügel froh? Was schwermütig? Jetzt sind wir aber tief im Schilf gelandet, denn die Frohbühlstrasse geht zurück auf den Flurnamen Frohloch, der wiederum fragen lässt, was ein Loch froh macht? Schon wieder falsch, weil Frohloch von Foloch kommt und Fuchsloch bedeutet.
Und was ist mit den Altstettern, die an die Herrligstrasse gezogen sind, damit es ihnen herrlich geht? Die Wahrheit ist bitter: Herrlig meint den alten Heerweg.

10.2 Morphologische Typologien

10.2.1 Morphologischer Sprachbau

Isolierender Sprachbau

Definition 10.2.1. Sprachen mit *isolierendem Sprachbau* haben keine gebundenen Morphe (Derivationsaffixe, Flexionsendungen).

- *Komposition* ist die einzige morphologische Operation.
- *Grammatische Beziehungen* und "morphologische" Merkmale werden durch selbständige Wörter und durch Wortstellungsregularitäten ausgedrückt.
- Mandarin-Chinesisch oder Vietnamesisch gelten als typische Vertreter dieses Sprachbaus.

Flektierender Sprachbau

Definition 10.2.2 (auch fusionierender Sprachbau)**.** Sprachen mit *flektierendem Sprachbau* haben viele gebundene Morphe, welche komplexe morphologische Merkmalswerte ausdrücken, und eine enge Verbindung mit den Kernmorphen eingehen.

- *Grammatische Beziehungen* und "morphologische" Merkmale werden durch Affixe ausgedrückt.
- Die Form eines gebundenen Morphems ist stark abhängig vom Stamm.
- Einzelne Laute können komplexe Information kodieren (Portmanteau-Morph): Das -o in Spanisch "hablo" (ich spreche) drückt 1. Person Singular Präsens Indikativ aus.
- Identische Morphe drücken unterschiedliche Funktionen aus (*Synkretismus*): Das -en in deutschen Verbformen kodiert Infinitiv, 1. oder 3. Person Plural.
- Latein oder Deutsch gelten als typische Vertreter dieses Sprachbaus.

Agglutinierender Sprachbau

Definition 10.2.3. Sprachen mit *agglutinierendem Sprachbau* verketten gebundene Morpheme hintereinander, welche jeweils eindeutig ein einzelnes Merkmal ausdrücken.

- Türkisch oder Finnisch gelten als typische Vertreter dieses Sprachbaus.
- Beide Sprachen haben Vokalharmonie, d.h. die Vokalisierung eines gebundenen Morphems ist abhängig vom letzten Stammvokal.

Beispiel 10.2.4 (Türkisch "evlerimin" = "meiner Häuser").
Siehe Tabelle 10.1 auf Seite 116.

Haus	Plural	mein	Genitiv
ev	ler	im	in

Tabelle 10.1: Agglutination im Türkischen

Das e in "ler", das i in "im" und das i in "in" entsteht durch Vokalharmonie. Weitere Information z.B. unter `http://www.tuerkisch-trainer.de/Sprachkurs/Grundlagen.htm`.

Polysynthetischer Sprachbau

Definition 10.2.5. Sprachen mit *polysynthetischem Sprachbau* verketten *gebundene und freie* Morpheme hintereinander zu langen Wörtern.

- Inuit-Sprachen gelten als typische Vertreter dieses Sprachbaus.

Beispiel 10.2.6 (Inuktitut). Siehe Tabelle 10.2 auf Seite 116.

Paris+mut+nngau+juma+niraq+lauq+sima+nngit+junga
Paris Terminalis-Kasus Weg-nach wollen sagen-dass Vergangenheit Perfektiv Zustand Negativ 1sg-intransitiv
Parimunngaujumaniralauqsimanngittunga
Ich sagte niemals, dass ich nach Paris gehen will.

Tabelle 10.2: Polysynthetisches Wort nach [Beesley und Karttunen 2003b, 376]

Fazit: Morphologischer Sprachbau

- Die *klassische morphologische Sprachtypologie* wurde zwischen 1818 (A.W. Schlegel) und 1836 (W. V. Humboldt) entwickelt.
- Die Klassifikationskriterien vermischen teilweise *Form*, d.h. die morphologischen Mittel, und *Funktion*, d.h. grammatische bzw. lexikalische Bedeutung.
- Die *moderne Universalienforschung* der Linguistik hat die klassische Konzeption kritisiert und die notwendigen begrifflichen Unterscheidungen herausgearbeitet.
- Echte Sprachen realisieren die morphologischen Typen *nicht in Reinform*. Je nach Phänomen sind die Typen unterschiedlich stark repräsentiert. Sogar Mandarin-Chinesisch kennt einige wenige Affixe.

- Deutsch hat sowohl flektierende, agglutinierende und polysynthetische Züge. Bei welchen Phänomenen?

  Flektierender Bau (selbstverständlich) bei der Flexion von Nomen, Adjektiven und Verben. Komparation der Adjektive ist agglutinierend. Bei der schwachen Verbflexion lässt sich ein Zeit-Modus-Suffix und ein Numerus-Person-Suffix isolieren, welches in dieser Reihenfolge agglutinierend erscheinen muss. Die Kompositabildung im Deutschen ist polysynthetisch, da Komposita mehr als ein freies Morphem ausdrücken können.

10.2.2 Morphologische Prozesse

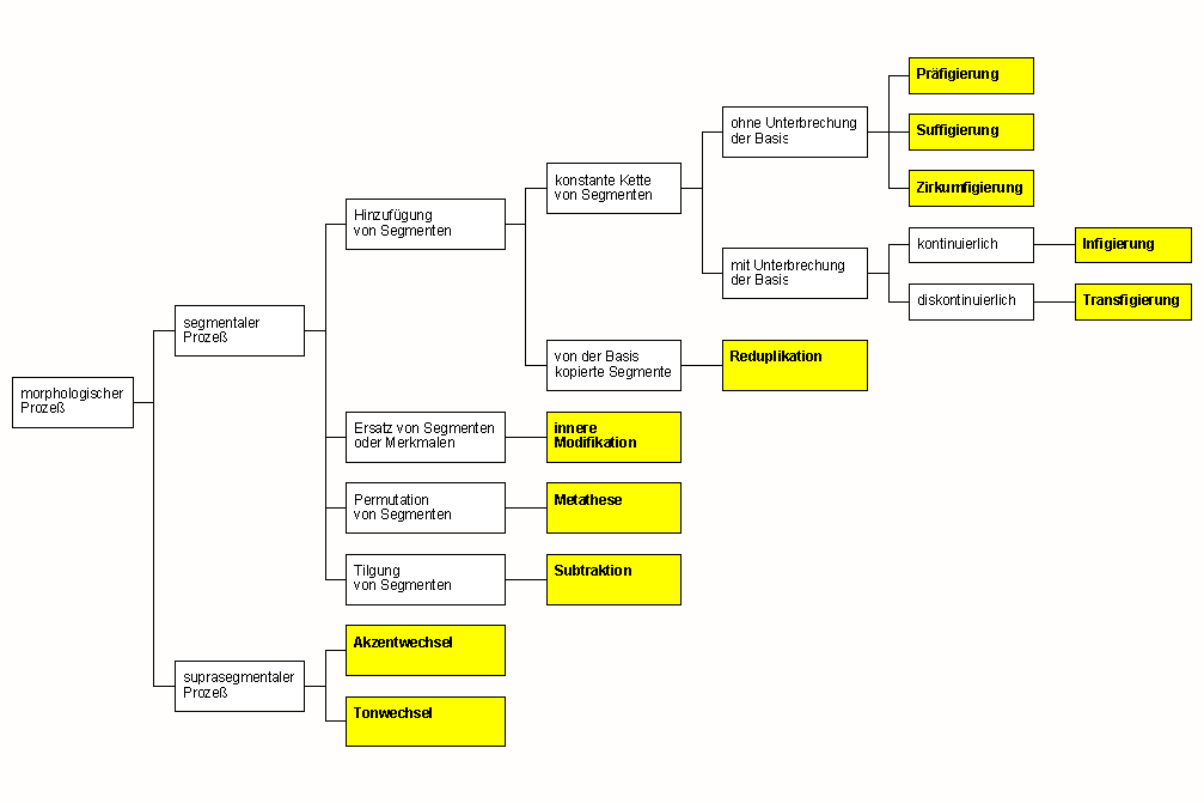


Abbildung 10.2: Morphologische Prozesse nach [Lehmann 2007]

Segmental vs. Suprasegmental

Definition 10.2.7 (Suprasegementale Merkmal nach [Glück 2000])**.** "Lautübergreifende bzw. sich nicht auf die sequentielle Abfolge von Segmenten beziehende Merkmale lautsprachl. Äußerungen, die sich signalphonet. im Grundfrequenz- und Intensitätsverlauf sowie der temporalen Ausprägung einzelner Segmente und Pausen äußern."

Akzentwechsel

engl. prodúce (Verb) vs. próduce (Nomen), contáct (Verb) vs. cóntact (Nomen).

Affigierung

Definition 10.2.8 (Affix)**.** Ein *Affix* ist ein gebundenes segmentales Morphem, das zur Bildung einer derivierten oder flektierten Form einer Basis (Wurzel oder Stamm) hinzugefügt wird.

Einteilung der Affixe nach Stellung

1. *Präfigierung*: Ein *Präfix* steht vor der Basis. (*Ge*-schrei)
2. *Suffigierung*: Ein *Suffix* folgt der Basis. (schrei-*en*)
3. *Zirkumfigierung*: Ein *Zirkumfix* umfasst die Basis. (*ge*-schrie-*en*)
4. *Infigierung*: Ein *Infix* unterbricht die (Wurzel-)Basis. (lat. vi-*n*-cere; vgl. Perfekt "vici")
5. *Transfigierung*: Ein unterbrochenes Affix unterbricht eine Basis an mindestens einer Stelle.

Beispiel: Transfigierung im Arabischen
Siehe Abb. 10.3 auf Seite 118.

Bedeutung	Mann	Haus	Tafel
Wurzel	*rʝl*	*bjt*	*lwħ*
Singular	*raʝul*	*bajt*	*lawħ*
Plural	*riʝāl*	*bujūt*	*ʔa-lwāħ*
Muster	$K_1iK_2āK_3$	$K_1uK_2ūK_3$	$ʔa\text{-}K_1K_2āK_3$

Abbildung 10.3: Pluralbildung im Arabischen nach [Lehmann 2007]

Reduplikation

Definition 10.2.9. Bei der *Reduplikation* wird eine Wurzel oder ein Stamm ganz oder teilweise (anlautende Silbe) verdoppelt. Lautliche Veränderungen sind möglich.

Beispiel 10.2.10 (Teilreduplikation und Vollreduplikation).

- Lateinisch: pello ("treibe") - pepuli ("trieb")
- Yukatekisch: tusah ("log") - tu'tusah ("log unverschämt"); xupah ("verschwendete") - xu'xupah ("verschwendete sinnlos")
- Deutsch: tagtäglich
- Vollreduplikation: Sumerisch: kur ("Land") - kurkur ("Länder")
- Welche Funktion scheint die derivationelle Reduplikation auf Grund der Beispiele zu kodieren?

Innere Modifikation/Abwandlung (Substitution)

Definition 10.2.11 (auch innere Abwandlung oder Substitution)**.** Bei der *inneren Modifikation* wird in der Basis ein Element oder Merkmal durch ein gleichartiges ersetzt.

Beispiel 10.2.12 (Ablaut (*apophony*)).

- Deutsch: s*i*nge - s*a*ng - ges*u*ngen

Beispiel 10.2.13 (Umlaut (engl. *umlaut*)).

- Deutsch: G*a*st - G*ä*ste

Suppletion

Beispiel 10.2.14.

- Lateinisch: ferro - tuli - latum ("tragen")
- Deutsch: bin - ist - sind - war etc.
- Deutsch: gut - besser - am besten

Definition 10.2.15. Bei der *Suppletion* wird ein Stamm (in einem Flexionsparadigma) vollständig durch einen anderen ersetzt.
Suppletivformen findet man typischerweise bei *hochfrequenten Lemmata*. Suppletion kann als extreme Form der Substitution (innere Modifikation) betrachtet werden.

Subtraktion

Definition 10.2.16. Bei der *Subtraktion* wird eine Basis gekürzt.

Beispiel 10.2.17 (Subtraktion bei französischen Adjektiven).
Bildung des maskulinen Genus durch Tilgung des auslautenden Stammkonsonanten. Graphematisch könnte es sich auch um e-Suffigierung an das Maskulinum handeln. Warum lautlich nicht?

maskulinum		femininum	
Schrift	**Laut**	**Schrift**	**Laut**
grand	[gʁɑ̃]	*grande*	[gʁɑ̃d]
petit	[pti]	*petite*	[ptit]
gris	[gʁi]	*grise*	[gʁiz]
gentil	[ʒɑ̃ti]	*gentille*	[ʒɑ̃tij]

Abbildung 10.4: Subtraktion bei französischen Adjektiven nach [Lehmann 2007]

10.3 Vertiefung

- Abschnitt "Morphology" [Trost 2003]
- Skript von Christian Lehmann [Lehmann 2007]: Morphologie und Syntax, dem Vieles aus dem Abschnitt "Morphologische Prozesse" entnommen ist.
- Abschnitt Morphologie in [Carstensen et al. 2004]

Kapitel 11

Morphologisches Engineering

Lernziele

- Kenntnis über die zentralen Fragen, welche bei einem Morphologieprojekt zu bedenken und entscheiden sind.
- Kenntnis von Techniken und Hilfsmitteln bei der Entwicklung und beim Testen

11.1 Planung von morphologischen Systemen

Planungsfragen

Anwendungsszenarien

- Wozu soll das morphologische System dienen?
- Was gibt es in diesem Bereich bereits? Was muss neu erschaffen werden?
- Welche urheber- und lizenzrechtlichen Beschränkungen bestehen für welche Einsatzbereiche (akademisch, kommerziell)?

Linguistische Fragen

- Welche Bereiche der Sprache sollen abgebildet werden?
- Rechtschreibekonventionen, regionale/dialektale Varianten vs. Standardsprache, Abdeckung von morphologischer Produktivität, ...
- Liegen Ressourcen vor, welche eingebunden werden können?

Planungsfragen

Software-technische Fragen

- Wie wird das Gesamtsystem aufgebaut? Architektur: Module und ihr Zusammenspiel
- Welcher Ansatz wird verfolgt?
- Welche Werkzeuge und Programmiertechniken sind geeignet/erforderlich für die Sprache X?
- Auswahl der Implementationstools und -techniken: `lexc`, `xfst`, `twolc`, eigene Skripten, ...; Komposition, Flag-Diacritics, fortgeschrittene Konstrukte (*compile-replace*, *priority union*)

11.2 Entwicklung

11.2.1 Tools

Nutzen von klassischen Software-Engineering-Tools

Versionskontrolle: Z.B. CVS , SVN, git, Bazaar

- *Zentrales Archiv* des Quellkodes; Entwickler arbeiten immer in privaten "<Sandkasten">, wo sie weiter entwickeln. Modifikationen werden ins Archiv zurückgespielt, wenn die Sandkastenversion zufriedenstellend funktioniert. git/Bazaar erlaubt mehrere verteilte Archive.
- Ermöglicht kontrollierte System-Entwicklung im *Team*: Entdeckung und Verwaltung von konkurrenzierenden Modifikationen. Einfache Übernahme von aktualisierten Dateien aus dem Archiv.
- Zuverlässiges *Rekonstruieren* von älteren Zuständen des Systems aus den verschiedenen Versionen der Dateien für Systemvergleiche.
- Erlaubt verschiedene *Releases* und verzweigende *Entwicklungsäste* über einen gemeinsamen Bestand von Quellkodes.
- Änderungen können dank der Unterstützung durch das Revisionssystem leicht zurückverfolgt werden.

Nutzen von klassischen Software-Engineering-Tools

Build-Unterstützung: Z.B. GNU make, ant, cons

- In einem "<Makefile"> werden die *Abhängigkeiten der Systemkomponenten* beschrieben.
- Und festgehalten, wie *neue Komponenten automatisch* aus bestehenden *generiert* werden.
- Bei Änderungen werden nur die von der modifizierten Komponente/Datei *abhängigen Komponenten* neu erzeugt.
- Durch Analysieren aller Abhängigkeiten kann ein einziger make-Befehl *zuverlässig und effizient* das ganze System neu erzeugen.
- *Unterschiedliche Varianten* eines Systems lassen sich gut in Makefiles definieren und erzeugen.

Makefile-Beispiel

make-Regel im Deutsch-Morphologieprojekt für Binärkompilation

```
# In Datei Makefile gespeicherte Regel:
%.xfst.bin : %.xfst                                          # Abhängigkeit
    xfst -e 'source $<' -e 'save defined $@' -stop           # Shell-Befehl
```

Falls eine Komponente eine Datei mit der Endung `.xfst.bin` (sog. *target*) *benötigt*: Prüfe, ob eine Datei ohne Endung `.bin` (sog. *prerequisite*) existiert. Prüfe (rekursiv) die Abhängigkeiten der *prerequisite* und *generiere* sie allenfalls neu. Falls das *target* älter ist als die *prerequisite*, *führe* den entsprechenden UNIX-Shell-Befehl *aus* (`$@` ist Make-Platzhalter für *target* und `$<` für *prerequisite*).

Beispiel 11.2.1 (Aufruf von `make`).

```
$ make adj.xfst.bin                # Erzeuge Ziel adj.xfst.bin
                                   # indem folgender Befehl ausgeführt wird
xfst -e 'source adj.xfst' -e 'save defined adj.xfst.bin' -stop
```

11.2.2 Inhalt

Linguistische Analyse und Planung

Formale linguistische Analyse
Ein Morphologiesystem benötigt eine möglichst gründliche vorangehende linguistische Analyse, welche die *Klassen, Regeln und Ausnahmen* möglichst detailliert zur Verfügung stellt.

Abstraktion und systematisches Wissen
"The ability to speak multiple languages, though admirable, doesn't make one a formal linguist any more than having a heart makes one a cardiologist."
[Beesley und Karttunen 2003b, 283]

Unterspezifikation linguistischer Beschreibung
Bei der Implementation handgeschriebener Regelsysteme ergeben sich immer *unvorhergesehene Überraschungen* trotz aller *Planung*: Unsicherheiten, Ungenauigkeiten, Unvollständigkeiten. Dies ist auch bei handgeschriebenen wissenschaftlichen Grammatiken der Fall.

Linguistische Analyse

Repräsentation der lexikalischen Seite
Die exakte *Form der Lemmata* ist eine *Design-Entscheidung* des Lexikographen.

- Systeme sollen sich möglichst an die *bestehenden lexikographischen Konventionen* einer bestimmten Sprache halten: Infinitiv als Grundform im Spanischen ("cantar"); aber in Latein z.B. die 1. Person Singular Indikativ Präsens ("canto"); abstrakte Stämme in Esperanto oder Sanskrit; Konsonantenwurzel in semitischen Sprachen
- Grund für diese konservative Einstellung: Bestehende lexikalische Ressourcen können so einfacher benutzt werden.
- Falls sich implementationstechnisch andere Grundformen eigentlich besser eignen, lassen sich die konventionellen oft auf der finalen lexikalischen Seite daraus erzeugen.
- Die Grundform flektierter Funktionswörter ("die", "sie") ist meist unklar.

Linguistische Analyse

Repräsentation der lexikalischen Seite
Die *Anzahl, Form und Abfolge von morphologischen Tags* ist eine *Design-Entscheidung* des Lexikographen.

- Die *lexikalische Grammatik*, d.h. Kombinationen und Abfolgen der Tags muss fest definiert sein. Ansonsten ist die Generierung von Wortformen kaum möglich.

- Typischerweise folgen die Tags dem Lemma – sprachspezifische Ausnahmen von dieser Regel sind möglich (semitische Sprachen oder Sprachen, wo Präfigierung wichtiger ist als Suffigierung).
- Namen sind Schall und Rauch: Ein Tag lässt sich mittels Ersetzungsoperator jederzeit umbenennen.
- Umstellungen der Tag-Reihenfolge sind allerdings etwas umständlicher – Benutzung von Flag-Diacritics kann hier helfen.

Empfohlene Tags nach Xerox-Konvention
Im Anhang G [Beesley und Karttunen 2003a], welcher im gedruckten Buch keinen Platz mehr fand, sind diverse sprachübergreifende morphologische Tags mit ihrer linguistischen Bedeutung aufgeführt.

Beispiel 11.2.2 (Xerox-Mehrzeichensymbol-Tags im +-Format).

```
+Noun    ! noun(house)
+Prop    ! proper noun(John)
+Art     ! article (like English the and a)
+Det     ! determiner (like this,that,those)
+Dig     ! digit-based word
+Aug     ! augmentative
+Dim     ! diminutive
...
```

Die Laufzeit-Applikation `lookup` zur morphologischen Analyse behandelt Tags mit der +-Konvention defaultmässig effizient als Mehrzeichensymbole.

11.2.3 Applikationen

Ein Kernsystem mit mehreren Runtime-Applikationen

Gemeinsames Kernsystem

- Das *Kernsystem* sollte möglichst viele Anwendungen unterstützen:
- Dialektale und Standard-*Varianten* (brasilianisches vs. kontinentales Portugiesisch), verschiedene Standards von Rechtschreibung, Umschreibungen (ss für ß, oe für ö)
- Konsequenz: Das Kernsystem soll so differenziert wie möglich sein (z.B. Merkmale für Brasilianisch oder rechtschreibereformbetroffene Wörter)

Runtime-Anwendungen

- Mache das Kernsystem *toleranter* (z.B. KAPITALISIERUNG oder kleinschreibung erlauben), *spezifischer* (Spellchecker für brasilianisches Portugiesisch), *erweitert* (Zufügen von Spezialwörtern), ...
- Erzeuge andere Ausgabe-Formate!

Runtime-Anwendungen

Beispiel 11.2.3 (Modifikation von Kern-System durch Ersetzungsregeln).

```
read regex StrictGermanCoreNetwork
.o. [ü (->) u e ] .o. [ö (->) o e ]
.o. [ä (->) a e ] .o. [ß (->) s s ];
```

Relaxierung von Kern-Systemen zur Laufzeit

Das `lookup/flookup`-Tool erlaubt morphologische Analysestrategien, welche *fehlerorientiert relaxieren* und zur Laufzeit die Komposition von Transduktoren simulieren:

- Falls eine Wortform nicht erkannt wird vom normalen System, erlaube deakzentuierten Input.
- Falls weiterhin keine Analyse vorliegt, erlaube Kapitalisierung.
- Falls weiterhin keine Analyse vorliegt, verwende einen heuristischen "Guesser", welcher ohne Lexikon arbeitet.

Klein-/Grossschreibung

Beispiel 11.2.4 (Grossschreibung).
Wie kann man Grossschreibung von normalerweise kleingeschriebenen Wörtern erlauben?

Beispiel 11.2.5 (Kapitalisierung).
Wie kann man durchgängige Grossschreibung erlauben wie in "Dieses Verhalten ist KRANK!!!"?

lookup-Strategien mit "virtueller" Komposition

Alternativ-Analysen, nur falls bisher keine gefunden

Analysen mit Relaxierung sollen nur dann gemacht werden, wenn keine "normale" Analyse möglich ist. [Beesley und Karttunen 2003b, 433]

1. *FST-Variablen*:

   ```
   deaccent     allow-deaccentuation.fst
   normalize    allow-allcaps.fst
   analyser     deutsch.fst
   ```

2. *Virtuelle Komposition*: Schreibe die FST-Variablen hintereinander!

3. *Strategien*

   ```
   analyser                        # Versuche zuerst normale Analyse
   allcap analyser                 # Sonst entkapitalisiere zuerst
   deaccent allcap analyser        # Sonst deakzentiere vor Entkapitalisierung
   ```

Das foma-Lookuptool erlaubt auf der Kommandozeile mehrere FST-Dateien, welche alternativ (`-a` Flag) nacheinander benutzt werden.

11.3 Testen

11.3.1 Fehlertypen

Typische Fehlertypen

Fehlende Analysen (*sin of omission*)

- Möglicher Grund I: Das System kennt das zugrundeliegende Lemma gar nicht: *lexikalische Lücke*.
- Möglicher Grund II: Das System hat Fehler im Lexikon oder den Regeln und liefert gewisse Analysen/Wortformen nicht (*partial analysis/generation failure*): *Linguistischer Fehler* im System.

Falsche Analysen (*sin of commission*)
Das System erzeugt falsche Wortformen und/oder falsche Analysen: Übergenerierung bzw. Überanalyse.
Welche Gründe kommen in Frage?

11.3.2 Testmethoden

Testmethoden

Manuelles Testen
Während dem Entwickeln werden in der Entwicklungsumgebung kleine Anfragen mit `apply` gemacht.
Kleine Experimente dienen zur Verifikation von Ideen.

Automatisches Testen
Systematisches Testen auf *Vollständigkeit* bzw. hohe Abdeckung und *Korrektheit* mit Hilfe der Laufzeit-Applikation (`lookup`) oder `xfst` und lexikalischen Testressourcen.

- *Reine Text-Korpora* helfen beim Verbessern der Abdeckung, indem fehlende Analysen von häufigen Wortformen ergänzt werden.
- *Manuell validierte Analyseresultate* sind die einzige Möglichkeit, die Korrektheit zu verbessern. Validiertes (und korrigiertes) Material kann für automatische *Regressions-Tests* im Entwicklungsprozess eingesetzt werden.

Regressions-Test

Definition 11.3.1 (engl. regression test)**.** Ein *Regressionstest* ist in der Softwareentwicklung ein Durchlauf von Testfällen durch ein System und einem Vergleich mit abgelegten korrekten Resultaten. Man bezweckt damit das Aufspüren von Nebenwirkungen, welche auf Grund von Modifikationen (Pflege, Änderung, Korrektur) von Software allenfalls eingeführt worden sind. Regressionstests werden meist vollständig automatisiert durchgeführt und evaluiert.

Abdeckungstestszenario im UNIX-Stil ►►►

1. Testkorpus nehmen
2. Simples Tokenisieren zu vertikalisiertem Text
3. Morphologische Analyse machen
4. Unanalysierbare Wortformen filtern
5. Nur Wortform in 1. Spalte ausgeben
6. Alphabetisches Sortieren
7. Format Häufigkeit-Wortform erzeugen
8. Numerisches Sortieren (Häufigstes zuoberst)

```
$ cat mycorpus.txt | \
  tr -sc "[:alpha:]" "[\n*]" | \
  lookup mylanguage.fst | \
  grep '+?' | \
  gawk '{print $1}' | \
  sort | \
  uniq -c | \
  sort -rnb > failures.sorted
```

Hinweis: Das Symbol `|` in der UNIX-Shell lenkt den Output eines Werkzeugs um als Input des nachfolgenden Werkzeugs – so braucht es keine Zwischendateien. Der `\` an Zeilenenden erlaubt einen langen Befehl auf mehreren Zeilen einzugeben.

Weitere Test- und Optimierungsmethoden

- Automatisches *Überprüfen* des Alphabets und der lexikalischen Grammatik mit Hilfe von xfst
- *Abgleich* mit den *Analysen* von anderen Morphologiesystemen (auch von früheren Versionen desselben Systems)
- *Abgleich* des Lexikons mit anderen lexikalischen Ressourcen auf der Ebene der *Lemmata*
- *Häufigkeitstest* über grossen Korpora (z.B. Web) mit Hilfe von Suchmaschinen-Anfragen
- ...

11.4 Vertiefung

- Die Kapitel 5, 6 und 9 geben viele praktische Hinweise, wie man erfolgreiche Projekte plant, entwickelt und testet.
- Revisionskontrollsysteme: Übersicht in Wikipedia `http://en.wikipedia.org/wiki/Comparison_of_revision_control_software`. Bazaar ist speziell auf einfache Benutzung entwickelt worden. git kann etwas widerspenstig sein, aber mit `github.com` und weiteren web-basierten OS-Repository können auch Anfänger starten (gute Tutorials).

- Automatisierte Build-Systeme: Übersicht in Wikipedia `http://en.wikipedia.org/wiki/List_of_build_automation_software`. Neben dem Klassiker Gnu make ist ant und scons verbreitet.
- Alternative Finite-State-Tools:
  - G. van Noords mächtige und prologbasierte *FSA*-Utilities von : `http://www.let.rug.nl/~vannoord/Fsa/`
  - Helmut Schmids schlanke, aber effiziente *SFST*: `http://www.ims.uni-stuttgart.de/projekte/gramotron/SOFTWARE/SFST.html`
  - C^{++}-basierte *OpenFst* Library für gewichtete Automaten: `http://www.openfst.org`
  - C^{++}-basierte Bibliothek *FSM2* von Th. Hanneforth `http://www.ling.uni-potsdam.de/~tom/`
  - Helsinki Finite-State Transducer Technology (HFST), welche foma und andere Tools integrieren: `http://www.ling.helsinki.fi/kieliteknologia/tutkimus/hfst/index.shtml`
  - ...

Kapitel 12

Projektideen für Schlussprojekt

12.1 Ziel

Schlussprojekt

- Ziel: Betreute, aber selbständige Durchführung eines kleinen *Schluss-Projekts* mit Finite-State-Methoden (25% der Schlussnote)
- Optionales Unterziel: Lernen, in kleinen Teams ein etwas komplexeres Projekt zu machen als das Übungsaufgabenniveau
- Aufwand: 16-24h. Die restliche Vorlesungszeit und Übungsstunden stehen insbesondere dafür zur Verfügung im Mai. D.h. Beratung und Betreuung von 14-17.30h.
- Postersessionmässige Kurzpräsentation (5 Minuten) des Projekts bzw. Projektstands am 27. Mai zur Vorlesungszeit (ab 14h)
- Späteste Abgabe: Mittwoch 26. Juni 2013 24h
- Bewertungskriterien: Korrektheit und Einfachheit der Implementation (50%); Qualität der Dokumentation (Quellkodekommentare/Readme) (30%); Präsentation: wurde das Wesentliche in der kurzen Zeit erwähnt (20%)

12.2 Themen

Morphologiesystem für Rumantsch
Reto Baumgartner als Koordinator und Integrator

- Entwicklung einer Morphologie für die flektierten Wortarten des Rumantsch.
- Detaillierte Grammatik `http://www.unifr.ch/rheto/documents/Gramminstr.pdf`
- Online-Lexikon (Deutsch-Rumantsch) `http://www.pledarigrond.ch`

molifde[1] erweitern

- Implementation von Numeralen (Ordinal-, Kardinalzahlen) mit STTS-Tagset sowie nominalisierte Ordinalzahlen (“Dritte”)

[1] http://kitt.cl.uzh.ch/kitt/molif/

- Implementation eines STTS-basierten Morphologiegenerators für Nominalkomposita mit unbekannten Erstgliedern. Gegeben ist ein STTS-Generator-Netzwerk für normale Nomina. `gemacht_ADJA_Pos.Fem.Gen.Sg.Sw` → `gemachten`
- Implementation eines STTS-basierten Guessers mit Hilfe von Priority-Union von häufigsten Suffix-Klassen. Gegeben ist ein STTS-basiertes Lexikon mit mehreren Tausenden Vollformen. Zweiteilig: (1) Erstellen der Statistiken für die häufigsten Stammendungen in Kombination mit Morphologieanalysetags. (2) Erstellen von xfst-Sourcen aus diesen Statistiken, welche einen Guesser ergeben.
- Flexionsklassen von molifde überprüfen auf linguistische Vollständigkeit und Redundanz (linguistisch) anhand von Alternativanalysen von GERTWOL und SMOR

Foma erweitern

- Idee 1: Implementation von Standardfunktionen (Gross-, Kleinschreibung, Buchstabenklassen) für foma ähnlich wie bei xfst (entweder in xfst selbst als Funktionen, Definitionen)
- Idee 2: Entwicklung von Regressionstests für verschiedene Operatoren und Funktionen von foma; d.h. verschiedene xfst-Skripte mit ihrem erwarteten Output erzeugen und automatische Testroutine (Make-basierte) machen

Finite-State Chunking (CoNLL-Task) für Englisch

- Als Shared-Task für mindestens 2 Teams
- Verfeinern der Regeln mit Kontext-Tests und Wort-Tests
- Statistiken über Chunk-Regeln auf Part-Of-Speech-Ebene werden zur Verfügung gestellt

Weiteres

- Termerkennung mit grossen Termresourcen aus dem Biomedizinischen Bereich
- Buchstaben-Aligner für Termerkennung mit vielen ähnlichen Termen mit kleinen orthographischen Unterschieden (Implementation nicht in xfst); Ziel: Minimierung von Termlexika
- Erstellen eine interessanten Übungsaufgabe für das nächste Semester inkl. Lösung

Eigene Ideen (!)

12.3 Abgabe

Minimalanforderungen für Abgabe

- Als Zip-Archiv oder als URL auf Source-Repository per Mail an `siclemat@cl.uzh.ch` bis spätestens 26. Juni
- readme.txt-Datei mit grundlegenden Angaben zum Projekt
- Gut kommentierter Quellkode

Checkliste für readme.txt

- Projekt-Ziel (3 Paragraphen),
- Implementationsstatus (was funktioniert, was ev. noch nicht)
- Ausbaumöglichkeiten (was könnte man noch machen)
- Anforderungen an die benötigte Software bzw. Betriebssystem
- Literaturhinweise und Quellen von benutzen Resourcen
- Befehle zum Erzeugen bzw. Laufenlassen, bzw. Testen des Systems

Literaturverzeichnis

[Antworth 1990] Antworth, Evan (1990). *PC-Kimmo: a two-level processor for morphological analysis*. Summer Institute of Linguistics, Dallas.

[Beesley 1998] Beesley, Kenneth (1998). *Constraining Separated Morphotactic Dependencies in Finite-State Grammars*, In: Karttunen, L. und K. Oflazer, Hrsg.: *Proceedings of FSMNLP'98: International Workshop on Finite-State Methods in Natural Language Processing*, S. 118–127. Bilkent University, `http://acl.ldc.upenn.edu/W/W98/W98-1312.pdf`.

[Beesley und Karttunen 2003a] Beesley, Kenneth R und L. Karttunen (2003a). *Appendix G: Recommended Analysis Tags*, electronic, http://www.cl.uzh.ch/siclemat/lehre/papers/BeesleyKarttunen2003a.pdf.

[Beesley und Karttunen 2003b] Beesley, Kenneth R. und L. Karttunen (2003b). *Finite-State Morphology: Xerox Tools and Techniques*. CSLI Publications.

[Beesley und Karttunen 2003c] Beesley, Kenneth R und L. Karttunen (2003c). *The twolc language*, electronic, `http://www.stanford.edu/~laurik/fsmbook/twolc.pdf`, http://www.cl.uzh.ch/siclemat/lehre/papers/BeesleyKarttunen2003twolc.pdf.

[Bussmann 2002] Bussmann, Hadumod (2002). *Lexikon der Sprachwissenschaft*. Kröner, Stuttgart, 3., aktual. und erw. Aufl.

[Carstensen et al. 2004] Carstensen, Kai-Uwe, C. Ebert, C. Endriss, S. Jekat, R. Klabunde und H. Langer, Hrsg. (2004). *Computerlinguistik und Sprachtechnologie : Eine Einführung*. Elsevier, München.

[Carstensen et al. 2009] Carstensen, Kai-Uwe, C. Ebert, C. Endriss, S. Jekat, R. Klabunde und H. Langer, Hrsg. (2009). *Computerlinguistik und Sprachtechnologie : Eine Einführung*. Spektrum, München.

[Chomsky und Halle 1968] Chomsky, Noam und M. Halle (1968). *The Sound Pattern of English*. Harper and Row, New York.

[Cohen-Sygal und Wintner 2005] Cohen-Sygal, Yael und S. Wintner (2005). *XFST2FSA: Comparing Two Finite-State Toolboxes*, In: *Proceedings of the ACL-2005 Workshop on Software*.

[Dudenredaktion 2005] Dudenredaktion, Hrsg. (2005). *Duden, die Grammatik: Unentbehrlich für richtiges Deutsch*, Bd. 4 d. Reihe *Der Duden*. Dudenverlag, 7. Aufl.

[Ehrhardt 2003] Ehrhardt, Jenny (2003). *Inuktitut: Affixliste*, `http://ling.kgw.tu-berlin.de/staff/Nowak/Affixliste.doc`, http://www.cl.uzh.ch/siclemat/lehre/papers/Ehrhardt2003.doc.

[Glück 2000] Glück, Helmut, Hrsg. (2000). *Metzler Lexikon Sprache*, Digitale Bibliothek Band 34. Directmedia, Berlin, Elektronische Ausgabe der zweiten, überarbeiteten und erweiterten Auflage Aufl.

[Grefenstette 1996] Grefenstette, Gregory (1996). *Light Parsing as Finite State Filtering*, In: *Workshop on extended finite state models of language*, Budapest, Hungary. ECAI'96, http://www.cl.uzh.ch/siclemat/lehre/papers/Grefenstette1996.pdf.

[Hausser 2001] Hausser, Roland (2001). *Foundations of computational linguistics manmachine communication in natural language*. Springer, Berlin, 2nd rev. and ext. Aufl.

[Hopcroft et al. 2002] Hopcroft, John E., R. Motwani und J. D. Ullman (2002). *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Pearson Studium, München, 2. überarbeitete Aufl.

[ICL 2011a] ICL (2011a). *Informationen zur Leistungsüberprüfung für Bachelorstudierende*, http://www.cl.uzh.ch/siclemat/lehre/papers/ICL2011.pdf.

[ICL 2011b] ICL (2011b). *Informationen zur Leistungsüberprüfung für Lizentiatsstudierende*, http://www.cl.uzh.ch/siclemat/lehre/papers/ICL2011a.pdf.

[Johnson 1972] Johnson, C. Douglas (1972). *Formal Aspects of Phonological Description*. Mouton, The Hague.

[Kaplan und Kay 1994] Kaplan, Ronald M. und M. Kay (1994). *Regular Models of Phonological Rule Systems*, Computational Linguistics, 20(3):331–378, `http://acl.ldc.upenn.edu/J/J94/J94-3001.pdf`.

[Karttunen 1995] Karttunen, Lauri (1995). *The Replace Operator*, In: *Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics*, S. 16–23, Cambridge, Mass. `http://www.aclweb.org/anthology/P95-1003.pdf`.

[Karttunen 1996] Karttunen, Lauri (1996). *Directed Replacement*, In: *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, S. 108–115, Santa Cruz, Ca. `http://www.aclweb.org/anthology/P96-1007.pdf`.

[Karttunen 2003] Karttunen, Lauri (2003). *Computing with realizational morphology*, In: *Computational Linguistics and Intelligent Text Processing*, S. 205–216. Springer.

[Karttunen und Beesley 2005] Karttunen, Lauri und K. R. Beesley (2005). *Twenty-five years of finite-state morphology*, In: Arppe, Antti, L. Carlson et al., Hrsg.: *Inquiries into Words, Constraints and Contexts (Festschrift in the Honour of Kimmo Koskenniemi on his 60th Birthday)*, S. 71–83. Gummerus Printing, Saarijärvi, Finland, `http://www2.parc.com/istl/members/karttune/publications/25years.pdf`.

[Koskeniemmi und Haapalainen 1996] Koskeniemmi, Kimmo und M. Haapalainen (1996). *GERTWOL – Lingsoft Oy*, In: Hausser, Roland, Hrsg.: *Linguistische Verifikation : Dokumentation zur Ersten Morpholympics 1994*, Nr. 34 in *Sprache und Information*, S. 121–140. Niemeyer, Tübingen, http://www.cl.uzh.ch/siclemat/lehre/papers/KoskeniemmiHaapalainen1996.pdf.

[Kunze 2007] Kunze, Jürgen (2007). *Vorlesung Morphologie, Die Grundbegriffe 1*, `http://www2.hu-berlin.de/compling/Lehrstuhl/Skripte/Morphologie/2_1.html`.

[Lakovic und Mai 1999] Lakovic, Zak und P. Mai (1999). *Über die Entstehung der endlichen Automaten oder Warum haben endliche Automaten keinen Namen?*, Entwicklung der Theorie der endlichen Automaten: Nervennetze von McCulloch und Pitts, Kleene und seine regulären Ereignisse und weitere Meilensteine., http://tal.cs.tu-berlin.de/gazzi/wise98/ausarbeitungen/ea.ps.gz, http://www.cl.uzh.ch/siclemat/lehre/papers/LakovicMai1999.pdf.

[Lehmann 2007] Lehmann, Christian (2007). *Morphologie und Syntax*, http://www.uni-erfurt.de/sprachwissenschaft/personal/lehmann/CL_Lehr/Morph&Syn/M&S_Index.html [cited Dienstag, 3. Juli 2007].

[Mohri 1997] Mohri, Mehryar (1997). *Finite-State Transducers in Language and Speech Processing*, Computational Linguistics, 23(2):269–311, http://acl.ldc.upenn.edu/J/J97/J97-2003.pdf.

[Mohri 2000] Mohri, Mehryar (2000). *Method and apparatus for compiling context-dependent rewrite rules and input strings - US Patent 6032111*, United States Patent: 6,032,111, http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.htm&r=1&f=G&l=50&s1=6,032,111.PN.

[Morgan und Stocker 2007] Morgan, Jennifer und T. Stocker (2007). *Ausschnitt aus Sternstunde Philosophie vom 18. 3. 2007 zum Thema 'Überlebensfrage Klima'*, http://www.sf.tv/sf1/sternstunden/index.php?docid=20070318, http://www.cl.uzh.ch/siclemat/lehre/papers/MorganStocker2007.mov.

[Nowak 2002] Nowak, Elke (2002). *Einführung ins Inuktitut*, http://ling.kgw.tu-berlin.de/staff/Nowak/Inumanu.doc, http://www.cl.uzh.ch/siclemat/lehre/papers/Nowak2002.doc.

[Roche und Schabes 1996] Roche, Emmanuel und Y. Schabes (1996). *Introduction to Finite-State Devices in Natural Language Processing*, MERL-TR-96-13, Mitsubishi Electric Research Laboratories, http://www.merl.com/papers/docs/TR96-13.pdf.

[Schützenberger 1961] Schützenberger, Marcel-Paul (1961). *A remark on finite transducers*, Information and Control, 4(2-3):185–196, http://dx.doi.org/10.1016/S0019-9958(61)80006-5.

[Stocker et al. 2004] Stocker, Christa, D. Macher, R. Studler, N. Bubenhofer, D. Crevlin, R. Liniger und M. Volk (2004). *Studien-CD Linguistik: Multimediale Einführungen und interaktive Übungen zur germanistischen Sprachwissenschaft*, Max Niemeyer Verlag, http://www.ds.uzh.ch/studien-cd.

[Stumpf 2001] Stumpf, Gregory Thomas (2001). *Inflectional Morphology: A Theory of Paradigm Structure*. Cambridge University Press.

[Tjong Kim Sang und Buchholz 2000] Tjong Kim Sang, Erik F. und S. Buchholz (2000). *Introduction to the CoNLL-2000 Shared Task: Chunking*, In: Cardie, Claire, W. Daelemans, C. Nedellec und E. Tjong Kim Sang, Hrsg.: *Proceedings of CoNLL-2000 and LLL-2000*, S. 127–132. Lisbon, Portugal.

[Trost 2003] Trost, Harald (2003). *Morphology*, In: Mitkov, Ruslan, Hrsg.: *Handbook of Computational Linguistics*, S. 25–47. Oxford University Press.

Index