Universität Bielefeld
Fakultät für Linguistik und Literaturwissenschaft
Modularbeit „Sprachverarbeitung“
Dr. Hans-Jürgen Eikmeyer
Sommersemester 2006

Finite State Morphology mit XFST und Perl

Nils Diewald
M.-Nr.: XXXXXXX
XXXXXXXXXXX XXX
XXXXX Bielefeld
nils.diewald@XXXXXXXXXXXXX.de

Inhaltsverzeichnis

1 Einleitung

Diese Hausarbeit behandelt die Erstellung eines morphologischen Lexikons in Form eines Finite State Transducers (FST) und die Interpretation des Lexikons in Perl. Ziel ist die Implementation eines morphologischen Analyseprogramms, das Texteingaben deutscher Sprache tokenweise analysiert und für jedes Token eine Menge erlaubter morphologischer Analysen zurückgibt.

Sowohl das im Rahmen der Arbeit mit dem *XEROX Finite State Tool* erstellte Lexikon als auch der Interpreter sind dabei Teil eines komplexeren Analysesystems, das auf Basis der hier vorgestellten Ausgabe eine weitergehende syntaktische Analyse durchführt. Dieses Syntax-Modul wurde von Daniel Jettka entwickelt.[1] Eine Online-Demonstration ist unter der Internet-Adresse `http://coli.lili.uni-bielefeld.de/cgi-bin/syntax-parser.pl`[2] zu finden.

Finite State Transducer

Finite State Transducer sind endliche Automaten mit übersetzenden Kanten. Sie akzeptieren nicht nur, wie alle endlichen Automaten, Symbolfolgen eines Eingabestrings, sondern geben für jedes Symbol ein Symbol zurück und erzeugen auf diese Weise für jeden akzeptierten Eingabestring mindestens einen Ausgabestring. Damit repräsentiert ein FST zwei Sprachen, die meist als linke und rechte Sprachen bezeichnet werden und sich auf die Seiten der übersetzenden Kanten beziehen. Ein FST lässt sich als Sixtupel $A =< \Sigma_1, \Sigma_2, \Phi, \delta, S, F >$ definieren:

Σ_1	Endliches Alphabet der linken Sprache
Σ_2	Endliches Alphabet der rechten Sprache
Φ	Endliche Menge aller Zustände/Knoten im Automaten
δ	Übergangsrelation, $\delta : \Phi \times \Sigma \rightarrow \Phi$
S	Startzustand/Wurzelknoten des FST, $S \in \Phi$
F	Menge aller finalen Zustände/Knoten des FST, $F \subseteq \Phi$

Σ_1 und Σ_2 enthalten dabei als spezielles Symbol das leere Zeichen ε. Auf diese Weise können Eingabe- und Ausgabestrings unterschiedlicher Länge sein, obgleich sie dem selben Pfad im Automaten folgen. Ein ε auf der Eingabeseite und ein Symbol

[1] Jettka (2008).
[2] Zuletzt abgerufen am 15.11.2008.

$x \neq \varepsilon$ auf der Ausgabeseite entspricht dabei einer Einfügung, ein Symbol $x \neq \varepsilon$ auf der Eingabeseite und ein Symbol ε auf der Ausgabeseite entspricht einer Löschung.

Automaten können sich hinsichtlich ihrer Determiniertheit unterscheiden: Deterministische endliche Automaten bieten für ein Eingabesymbol von einem Zustand genau eine Übergangsmöglichkeit an, d.h. je nach Eingabe ist der Folgezustand eindeutig. Nichtdeterministische Automaten erlauben mehrere mögliche Übergänge von einem Zustand über ein Eingabesymbol.

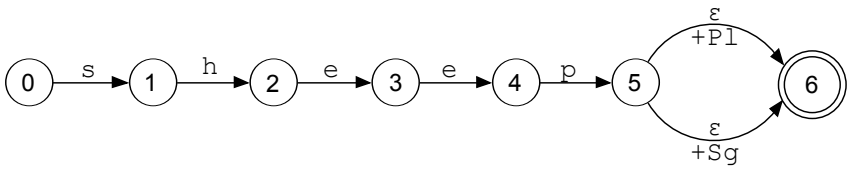

Abbildung 1: FST als Netz. Die Eingabe des Strings `sheep` gibt die beiden Analysestrings `sheep+Pl` und `sheep+Sg` zurück.

Finite State Transducer können in der Computerlinguistik zu vielerlei Zwecken eingesetzt werden, beispielsweise zur Erzeugung phonetischer Oberflächenformen aus Textwörtern in *Text-to-Speech*-Systemen. Im Falle eines morphologischen Lexikons, wie es in dieser Arbeit vorgestellt wird, würde eine Oberflächenform in einen oder mehrere Analysestrings[3] übersetzbar sein, die neben der Stammform des eingegebenen Wortes zahlreiche morphologische Informationen enthalten. Dieser Automat könnte sowohl zur Analyse als auch zur Generierung von Wortformen dienen. Hierfür genügt die Erzeugung eines Transducers, denn FSTs können nicht nur von einer akzeptierenden Sprache in eine generierende Sprache übersetzen – die beiden repräsentierten Sprachen lassen sich bidirektional übersetzen. Es muss lediglich vor der Eingabe des Inputstrings festgelegt sein, welche Sprache die akzeptierende ist.

Durch diese Bidirektionalität ist dieser Ansatz unidirektionalen Transformationssystemen, in denen kaskardierend auf einen Inputstring Ersetzungsregeln angewendet werden um einen Outputstring zu erhalten, überlegen.[4] Er unterscheidet sich vom kaskadierenden Ansatz in folgenden Punkten:[5]

- Die Regeln sind symbolbasiert und ihre Verarbeitung geschieht nicht sequentiell sondern parallel.

[3] S. Fitschen (2004), S. 14.
[4] S. z.B. für den Bereich des Stemmings das populäre System nach Porter (1980).
[5] Aus: Karttunen und Beesley (2001), S. 4f.

- Die Regeln können sowohl in Bezug auf den Analysekontext als auch den Oberflächenkontext sensitiv formuliert werden.
- Das Nachschlagen im Lexikon und die morphologische Analyse geschehen in einem Vorgang.

Koskenniemi, der diese Form der parallelen Transformation im Gegensatz zur sequentiellen Transformation Anfang der 80er Jahre für den Bereich der Morphologie entdeckte, bezeichnete sie als „two-level model“. Diese Bezeichnung sollte darauf deuten, dass außer der lexikalischen Form und der Oberflächenform keine Zwischenformen („intermediate levels“) während der Transformation existieren.[6]

Allerdings war die Formulierung einzelner, nacheinanderfolgender Regeln zu diesem Zeitpunkt noch ungleich einfacher als die Erstellung eines komplexen Transducers. Daher basierten die Anfänge der „Two-Level Morphology“ auch noch auf separaten Regelautomaten, die kein geschlossenes Lexikon ergaben,[7] bis schließlich ein Compiler entwickelt wurde, der durch die Komposition vieler einzelner Automaten ein Lexikon bestehend aus einem einzigen deterministischen und minimisierten Finite State Transducer erstellen konnte.[8] Der im *Xerox Finite State Tool* eingesetzte Compiler basiert auf diesem Prinzip.

2 Erstellung eines Lexikons als FST

Die Erstellung eines Lexikons in Form eines Transducers ist aufwändig und bedarf einer guten Software-Unterstützung bei der Beschreibung und der Kompilation. Das *Xerox Finite State Tool* bietet hierfür einen leistungsstarken Compiler und zwei verschiedene, sich in weiten Teilen ergänzende Beschreibungssprachen: *lexc* und *xfst*. Bei *lexc* handelt es sich um eine Beschreibungssprache für natürlichsprachliche Lexika, in der Einträge nebst ihrer Übersetzungen definiert werden. Wörter können in mehrere Elemente separiert werden, beispielsweise Morpheme, und durch Fortsetzungslexika erweitert werden, um etwa die Flexion eines Wortes durch verschiedene Endungen ein und desselben Stammes zu beschreiben.

[6] Koskenniemi (1983), S. 683.
[7] Koskenniemi implementierte seinen Ansatz zunächst als Trie-Forest, also als Trie (auch bekannt als *lexicographic* oder *prefix tree*), dessen Blätter zu Wurzeln weiterer Tries führten (Karttunen und Beesley, 2001).
[8] Karttunen und Beesley (1992).

xfst bietet die Möglichkeit, Transducer-Regeln ähnlich des „Two-Level“ Ersetzungsformalismus mittels eines umfangreichen Repertoires regulärer Ausdrücke zu definieren.

Im Folgenden soll anhand eines Beispiels die Verwendung von *lexc* und *xfst* zur Erstellung eines komplexen morphologischen Lexikons gezeigt werden.

Beispiel

lexc- und *xfst*-Dateien sind Textdateien und können entsprechend in beliebigen Editoren bearbeitet werden. Die hier vorgestellten Lexikon-Fragmente werden aufeinander aufbauend dargestellt. Das vollständige Lexikon nach dem letzten Erweiterungsschritt, ist in Anhang B zu finden.

Als erstes Beispiel wird das Wort „lachen“ mit seiner Konjugation in *lexc* beschrieben:

```
1  Multichar_Symbols
2    +1P +2P +3P
3    +Sg +Pl
4    +Pres
5
6  LEXICON Root
7    lach VerbEnd;
8
9  LEXICON VerbEnd
10   +1P+Sg+Pres:e   #;
11   +2P+Sg+Pres:st  #;
12   +3P+Sg+Pres:t   #;
13   +1P+Pl+Pres:en  #;
14   +2P+Pl+Pres:t   #;
15   +3P+Pl+Pres:en  #;
```

verb.lexc (1)

In einem Prolog werden hierbei zunächst die Mehrzeichensymbole der Analyse deklariert. Da FSTs symbolweise verarbeiten und *lexc* von einer zeichenweisen Beschreibung ausgeht, müssen Mehrzeichensymbole zunächst eingeführt werden, damit sie in den Lexika korrekt erkannt werden. Danach folgt das Wurzellexikon, das immer den eindeutigen Namen *Root*[9] hat. Dort wird der Stamm des Wortes „lachen“ eingeführt und dahinter auf das Fortsetzungslexikon *VerbEnd* verwiesen. Hier werden

[9] Alle weiteren Lexika dürfen beliebig benannt werden.

für die verschiedenen Kombinationen von Person und Numerus die Verbendungen im Präsens deklariert. Die Übersetzung wird durch einen Doppelpunkt dargestellt. Die linken Symbole übersetzen in die rechten Symbole (und vice versa). Hinter den Wortendungen wird auf das Fortsetzungslexikon # verwiesen. Dies ist das terminale Lexikon in *lexc* und bedeutet, dass das Wort vollständig deklariert ist. Im Transducer würde entsprechend auf einen terminalen Knoten verwiesen werden, wie in der kompilierten Darstellung als Netz in Abb. 2 zu sehen ist.

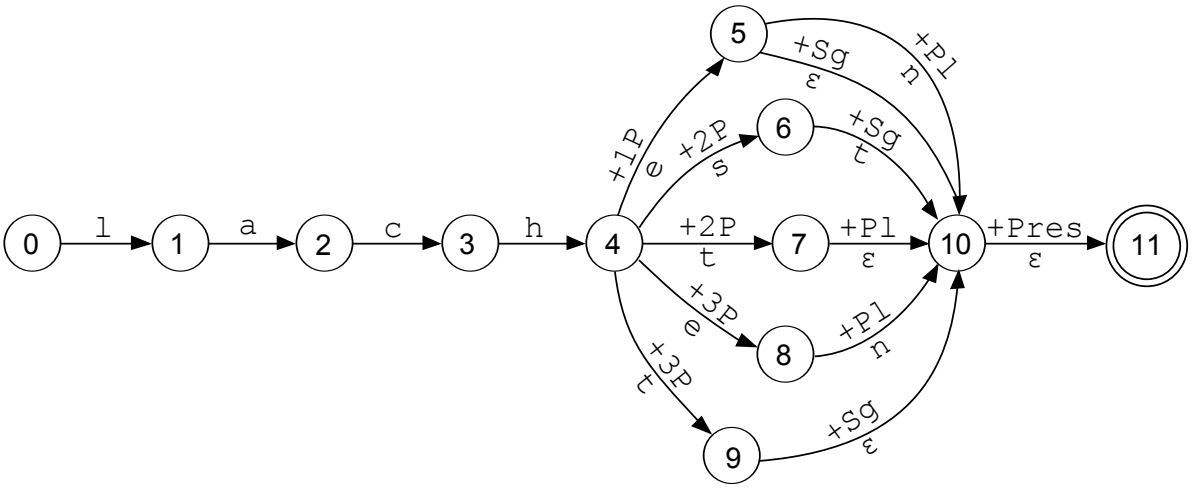

Abbildung 2: Darstellung des Lexikons „verb.lexc (1)“ als Netz. $S = 0$, Alle Knoten $x \in F$ sind mit Doppelkreis dargestellt.

Im nächsten Schritt wird das Wort „helfen“ dem Lexikon hinzugefügt.

```
1  Multichar_Symbols
2    (...)
3    +Pres +Imp
4    +Verb +Mark1 +Mark2
5
6  LEXICON Root
7    lach    VerbMarker1;
8    helf    VerbMarker2;
9
10 LEXICON VerbMarker1
11   +Verb:+Verb+Mark1 VerbEnd;
12
13 LEXICON VerbMarker2
14   +Verb:+Verb+Mark2 VerbEnd;
```

verb.lexc (2)

Das Wort „helfen“ verhält sich in der 2. und 3. Person Singular anders als „lachen“; es wird gebeugt. Um die Wortformen „*helfst“ und „*helft“ durch die korrekten Wortformen „hilfst“ und „hilft“ zu ersetzen, wird dem Wort ein zusätzliches Mehrzeichensymbol hinzugefügt – ein Marker – der diesen Beugungsbedarf für die weitere Verarbeitung im Transducer deutlich macht. Damit wird „helfen“ einer anderen Flexionsklasse zugeordnet als „lachen“.

Die Weiterverarbeitung des Transducers kann nun mit der Beschreibungssprache *xfst* geschehen. *xfst* ist eine Beschreibungssprache, die reguläre Ausdrücke für die Erstellung und Manipulation von Netzen benutzt und gut geeignet ist, Lexika, die mittels *lexc* erstellt wurden, hinsichtlich Flexion zu manipulieren.

```
1  read lexc < verb.lexc
2  def lexVerb;
3
4  def 2P3PSgPres [["+2P"|"+3P"] ?* "+Sg" ?* "+Pres"];
5
6  # Präsens:
7  def PresM2 [e -> i || _ ?* "+Mark2" ?* 2P3PSgPres ?*];
8
9  def Eraser [["+1P"|"+2P"|"+3P"|
10               "+Sg"|"+Pl"|
11               "+Pres"|"+Mark1"|"+Mark2"|
12               "+Verb"] -> 0];
13
14 clear stack;
15 read regex lexVerb .o. PresM2 .o. Eraser;
```

verb.xfst (1)

Zunächst wird hierbei die *lexc*-Datei geladen und als Netz *lexVerb* definiert. Zusätzlich wird ein Netz *2P3PSgPres* durch einen regulären Ausdruck definiert, der alle Einträge, die 2. oder 3. Person Singular Präsens sind, akzeptiert. Mit *PresM2* nun wird ein Transducer deklariert, der alle Einträge, die 2. oder 3. Person Singular Präsens sind und den Flexionsmarker *+Mark2* enthalten, dahingehend manipuliert, dass ein enthaltenes „e“ (z.B. in „*helfst“) in ein „i“ übersetzt wird („hilfst“). In Zeile 9ff wird ein weiterer Transducer *Eraser* definiert, der dafür sorgt, dass eine Seite des Transducers keine Mehrzeichensymbole enthält. In Zeile 14 wird der Speicher von allen erstellten Netzen befreit (dabei bleiben alle mit „def“ abgelegten Netze unter ihrem Namen erhalten) und ein neuer Eintrag abgelegt, der mittels des Kompositions-Operators („.o.“) alle Transducer nacheinander anwendet, d.h. *lexVerb* wird durch *PresM2* ma-

nipuliert und das Ergebnis durch *Eraser* bereinigt.

Im nächsten Schritt wird das Präteritum der Verbflexion hinzugefügt.

```
Multichar_Symbols
  (...)
  +Pres +Imp +Temp

LEXICON VerbEnd
  (...)

  +1P+Sg+Imp:+Temp+1P+Sg+Imp    #;
  +2P+Sg+Imp:+Tempst+2P+Sg+Imp  #;
  +3P+Sg+Imp:+Temp+3P+Sg+Imp    #;
  +1P+Pl+Imp:+Tempen+1P+Pl+Imp  #;
  +2P+Pl+Imp:+Tempt+2P+Pl+Imp   #;
  +3P+Pl+Imp:+Tempen+3P+Pl+Imp  #;
```

verb.lexc (3)

```
# Imperfekt:
def ImpM1 [["+Temp" (e)] -> {te} ||
           ?* "+Mark1" ?* _ \e ?* "+Imp" ?*];
def ImpM2 [e -> a || _ ?* "+Mark2" ?* "+Imp" ?*];
def Imp ImpM1 .o. ImpM2;

def Eraser [[(...) "+Pres"|"+Imp"|"+Perf"|"+Temp"] -> 0];

clear stack;
read regex (...) Imp .o. Eraser;
```

verb.xfst (2)

Auch hier sind erneut die verschiedenen Flexionsklassen von Bedeutung. Genügt bei dem Wort „lachen“ der Einschub eines „te“ an Stelle des Zeitformmarkers *+Temp* (Transducer „ImpM1“), findet bei „helfen“ erneut eine Beugung statt, da gilt: „Ich half“ aber „*Ich helfte“ (Transducer „ImpM2“).[10]

lexc bietet, ebenso wie *xfst*, die Möglichkeit reguläre Ausdrücke für die Definition von Übersetzungen zu deklarieren. Hierfür werden die Ausdrücke in spitze Klammern geschrieben. Am Beispiel der Partizip-Perfekt-Verbendung ist zu sehen, wie in einer Zeile alle Kombinationen definiert werden können.

[10] Um die Bildung von „*lachteen“ (1. und 3. Person Plural Imperfekt von „lachen“) durch das Einfügen von „te“ zu verhindern, wird ein eventuell vorkommendes „e“ von dem Transducer *ImpM1* mitübersetzt.

```
Multichar_Symbols
  (...)
  +Pres +Imp +Perf +Temp
  +Pref

LEXICON Root
  0:+Pref    Verbs;

LEXICON Verbs
  lach    VerbMarker1;
  helf    VerbMarker2;

LEXICON VerbEnd
  (...)
  < 0:t ["+1P"|"+2P"|"+3P"] ["+Sg"|"+Pl"] "+Perf" > #;
```

verb.lexc (4)

Hier wird zudem das Lexikon „Root“ verändert um vor dem Wortstamm einen Präfix-Marker zu setzen. Dieser Marker wird in der weiteren Verarbeitung durch *xfst* zu „ge“, falls der Perfekt-Marker gesetzt wurde.

```
# Perfekt:
def PerfEnd [ t -> {en} || "+Mark2" ?* _ ?* "+Perf" ?*];
def PerfM2 [e -> o || _ ?* "+Mark2" ?* "+Perf" ?*];
def Perf PerfEnd .o. PerfM2;

def gePrefix ["+Pref" -> {ge} || .#._ ?* "+Perf" ?*];

def Eraser [[(...) "+Pref"] -> 0];

read regex (...) Perf .o. gePrefix .o. Eraser;
```

verb.xfst (3)

Für eine produktivere Verwendung des Lexikons ist es möglich, untrennbare Verbpräfixe anzugeben, um etwa „verlachen“, „verhelfen“ oder „behelfen“ zu erlauben.

```
Multichar_Symbols
  (...)
  +Pref +NoPref

Definitions
  be  = {be} :["+NoPref" {BE} ];
```

```
7    ver = {ver}:["+NoPref" {VER}];
8
9    (...)
10
11   LEXICON Verbs
12     < [0|ver] {lach}>    VerbMarker1;
13     < [0|ver|be] {helf}>    VerbMarker2;
```

verb.lexc (5)

Die Präfixe werden im Prolog in einer Definitions-Liste angegeben. Hier sind Netzdefinitionen erlaubt, die später in den Lexika wieder aufgegriffen werden können. Auf diese Weise ist es möglich, sehr einfach viele untrennbare Verbpräfixe an einen Wortstamm zu binden, z.B. `<[ver|be|er|ent|ueber|unter|um|wider|durch|0]fahr>`.

```
1    def gePrefix ["+Pref" -> {ge} ||
2                  .#._ [? - "+NoPref"] ?* "+Perf" ?*];
3
4    def specialChars [E -> e, B -> b, V -> v, R -> r];
5
6    def Eraser [[(...) "+NoPref"] -> 0];
7
8    (...)
9
10   read regex (...) specialChars .o. Eraser;
```

verb.xfst (4)

Die Verwendung der „Sonderzeichen“ (Großbuchstaben) ist hier notwendig, um die Präfixe vor Beugungsregeln zu schützen, so dass „behalf“, aber nicht „*bahalf“ entstehen.

Im nächsten Schritt werden die Imperativformen durch das Hinzufügen von Modus-Markern eingeführt. Hierbei wird erneut die Möglichkeit regulärer Ausdrücke in *lexc* genutzt um alternative Wortformen zu erlauben, z.B. „Lach!“ ebenso wie „Lache!“.

```
1    Multichar_Symbols
2      (...)
3      +MInd +MImp +MKonj
4
5      (...)
6
7    LEXICON VerbEnd
8      (...)
```

```
9
10    < 0:[0|e] "+Sg" "+MImp" > #;
11    < 0:t "+P1" "+MImp" > #;
```

verb.lexc (6)

Durch eine Erweiterung der *PresM2*-Regel wird die e-i-Beugung beim Verb „helfen“ auch für den Imperativ ausgenutzt. Da die alternative Form „*Hilfe!“ im Sinne eines Imperativs jedoch unzulässig ist, wird für den Fall der +Mark2-Flexionsklasse die Endung unterbunden, indem das „e“ gelöscht wird.[11]

```
1   (...)
2
3   # Präsens/Imperativ:
4   def PresM2a [e -> I || _ ?* "+Mark2" ?* [2P3PSg|"+Sg" "+MImp"] ?*];
5   def PresM2b [e -> 0 || ?* I ?* "+Mark2" ?* _ "+Sg" "+MImp" ?*];
6   def PresM2 PresM2a .o. PresM2b .o. [I -> i];
7
8   (...)
9
10  def Eraser [[(...) "+MImp"] -> 0];
11
12  (...)
```

verb.xfst (5)

3 Interpretation des Transducers

Nach der Erstellung eines Lexikons können Nachschlageoperationen darauf angewandt werden. Liegt das Lexikon als FST vor, bedeutet jedes Nachschlagen gleichzeitig eine Übersetzung.

Ausgangspunkt für jedes Nachschlagen im Transducer ist der Startknoten S. Davon ausgehend werden gemäß des Eingabestrings und der als akzeptierend ausgewählten Sprachseite die Kanten des Netzes traversiert, d.h. begangen. Bei jedem Kantenübergang wird ein Symbol des Eingabestrings akzeptiert und ein Symbol in den Ausgabestring geschrieben. Ausnahmen bilden hierbei Kanten mit ε: Ist das akzeptierende Symbol einer Kante ε, wird kein Symbol des Inputstrings verarbeitet, die Kante

[11] Hierbei entsteht nicht, wie zu vermuten wäre, eine zweite Oberflächenform „hilf“ – Der Xerox Compiler sorgt durch eine Minimisierung dafür, das keine zwei Oberflächenformen mit identischen Übersetzungen im Lexikon existieren können.

ist dennoch traversierbar. Ist ε das Symbol der Ausgabeseite, wird kein Symbol in den Ausgabestring geschrieben.

Die Übersetzung eines Eingabestrings ist erfolgreich, wenn der Eingabestring vollständig verarbeitet wurde und ein finaler Zustand $x \in F$ erreicht wurde.

Ein Transducer kann für einen Eingabestring mehr als eine Übersetzung zurückliefern. Hierfür muss er jeden möglichen Pfad des Automaten traversieren, d.h. an jeder Gabelung des Netzes, an der mehrere mögliche Kantenübergänge existieren, müssen alle traversiert werden. Um dies zu erreichen, müssen mögliche Übergänge auf einem Stapel (*Stack*) gespeichert werden um zu einem späteren Zeitpunkt verarbeitet zu werden. Die im Folgekapitel vorgestellten Implementationen sehen für die Verarbeitung eine Tiefensuche vor.

Als Beispiel für eine solche Tiefensuche, soll die Übersetzung eines Wortes mit einem einfachen FST-Lexikon dienen.[12]

Beispiel

Das Lexikon enthält die Deklination des Wortes „Jäger“[13]. Die Beschreibung in *lexc* ist in Abb. 3 dargestellt, in Netzform in Abb. 3. Untersucht wird die Generierungsrichtung, d.h. als Eingabestring wird die Analysestring-Schreibform verwendet, als Ausgabestrings die gültigen Wortformen erwartet. Der Eingabestring lautet `J, a, e, g, e, r, +Mas, +Dat, +Pl`.

Es wird folgende Notation verwendet:

$pointer_{in}$: Zeiger auf die aktuelle Zeichen-Position des Input-Strings
$pointer_{out}$: Zeiger auf die aktuelle Zeichen-Position des Output-Strings
$label_{up}$: Akzeptor-Label einer Kante
$label_{down}$: Produzenten-Label einer Kante

```
1  Multichar_Symbols
2    +Sg +Pl +Nom +Gen +Dat +Acc +Mas
3
4  LEXICON Root
5    Jaeger Endung;
6
7  LEXICON Endung
8    +Mas+Nom+Sg:0   #;
```

[12] Der dargestellte Algorithmus entspricht der Implementation 1 des Kapitels 4.
[13] Wie das Lexikon in Kapitel 4.1 enthält das Beispiel-Lexikon keine Umlaute – der Buchstabe „ä“ wird als „ae“ umschrieben.

```
9     +Mas+Gen+Sg:s   #;
10    +Mas+Dat+Sg:0   #;
11    +Mas+Acc+Sg:0   #;
12    +Mas+Nom+Pl:0   #;
13    +Mas+Gen+Pl:0   #;
14    +Mas+Dat+Pl:n   #;
15    +Mas+Acc+Pl:0   #;
```

„Jäger“-Transducer in *lexc* definiert.

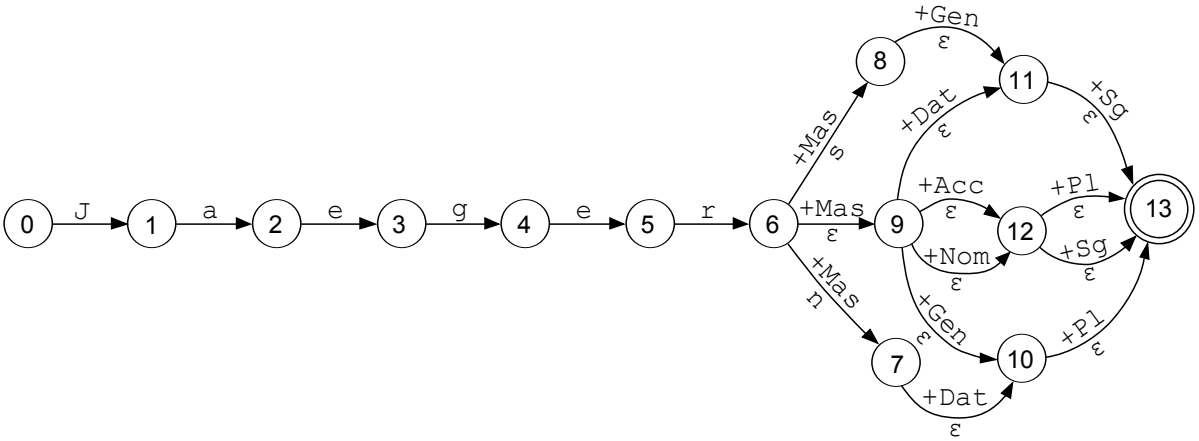

Abbildung 3: Darstellung des „Jaeger“-Transducers als Netz. $S = 0$, Zustände $x \in F$ sind mit Doppelkreis dargestellt.

Der Eingabestring wird symbolweise von links nach rechts gelesen. Parallel hierzu wird beginnend vom Startzustand des FSTs traversiert. Das erste Symbol des Eingabestrings J wird gelesen und überprüft, welche Zustandsübergänge von $S = 0$ das Symbol akzeptieren und zu einem Übergang führen können. Die Kante $0 \xrightarrow{J} 1$ akzeptiert das Zeichen J und kann traversiert werden. Das Symbolrepertoire von Xerox Finite-State-Transducern gibt verschiedene Regeln für Übergänge vor.

	$label_{up}$	$label_{down}$	$pointer_{in}$	$pointer_{out}$	$\curvearrowright$
1	x		++	++	x
2	ε				ε
3	?		++	++	?
4	x	y	++	++	y
5	x	ε	++		ε
6	ε	x		++	x

Unterschieden werden einfache und übersetzende Kanten. Der Übergang $0 \xrightarrow{J} 1$ ist eine einfache Kante vom Typ `1` mit $label_{up} = J$. Bei diesem Übergang wird das Symbol an Stelle $pointer_{in}$ im Inputstring akzeptiert und in den Ausgabestring an der Stelle $pointer_{out}$ geschrieben. Beide Ausgabezeiger werden danach inkrementiert, d.h. um 1 erhöht. Ein weiterer Typ eines einfachen Übergangs ist die Nullkante $\xrightarrow{\varepsilon}$. Diese Typ `2` Kante kann jederzeit traversiert werden, da sie keine Symbole akzeptiert oder produziert (entsprechend bleiben $pointer_{in}$ und $pointer_{out}$ unverändert) und lediglich zu einer Zustandsveränderung führt. Der dritte einfache Kantentyp $\xrightarrow{?}$ akzeptiert ein unbekanntes Symbol[14] an $pointer_{in}$ und gibt dieses an $pointer_{out}$ wieder aus.[15] Beide Zeiger werden in Folge inkrementiert.

Der Beispieleingabestring wird über Typ `1` Übergänge traversiert bis der Zustand 6 erreicht wurde und $pointer_{in}$ sowie $pointer_{out}$ an sechster Position des Eingabe- bzw. des Ausgabestrings steht. Das Eingabesymbol lautet `+Mas`. Von Knoten 6 gehen drei Kanten ab, die alle das Symbol `+Mas` akzeptieren. Der Übergang $6 \xrightarrow[n]{+Mas} 7$ ist vom Typ `4`, wobei ein Symbol an $pointer_{in}$ akzeptiert wird, ein anderes, im Fall des Übergangs von 6 zu 7 das Zeichen `n`, hingegen an $pointer_{out}$ in die Ausgabe geschrieben wird. Beide Zeiger werden danach inkrementiert. Der Übergang $6 \xrightarrow[s]{+Mas} 8$ ist ebenfalls vom Typ `4`. Die dritte Übergangsmöglichkeit $6 \xrightarrow[\varepsilon]{+Mas} 9$ ist vom Typ `5`, hierbei wird kein Zeichen in den Outputstring geschrieben und $pointer_{out}$, anders als $pointer_{in}$, nicht inkrementiert.

Für den nächsten Schritt werden die Übergänge $6 \xrightarrow[\varepsilon]{+Mas} 9$ und $6 \xrightarrow[s]{+Mas} 8$ auf einem Stapelspeicher gesichert, zusammen mit den aktuellen $pointer_{in}$- und $pointer_{out}$-Werten. Dieser Stapelspeicher dient der späteren Bearbeitung.

Danach wird der Übergang $6 \xrightarrow[n]{+Mas} 7$ traversiert, in den Outputstring wird an $pointer_{out}$ das Zeichen `n` geschrieben und beide Zeiger werden um eins erhöht. Im folgenden Schritt wird der Übergang $7 \xrightarrow[\varepsilon]{+Dat} 10$ vom Typ 5 traversiert, danach $10 \xrightarrow[\varepsilon]{+Pl} 13$, wonach ein terminaler Zustand erreicht wurde. Da in den letzten zwei Schritten gemäß Typ `5` $pointer_{out}$ nicht erhöht wurde, ist der aktuelle Status der Verarbeitung wie in Abb. 4 zu sehen.

[14] Das Fragezeichen bedeutet im Transducer-Netz ein beliebiges Symbol $x \notin \Sigma_1$ bzw. $x \notin \Sigma_2$. Damit hat es eine andere Bedeutung als in *xfst* oder *lexc*, wo es für ein beliebiges Zeichen unabhängig von Σ steht.

[15] In Kapitel 4 werden zwei verschiedene Implementationen vorgestellt, wobei nur die Listenvariante eine Unterscheidung zwischen einfachen und übersetzenden Kanten vornimmt. Alle einfachen Kanten können auch durch übersetzende Kanten formuliert werden; anstelle von $\xrightarrow{x}$ könnte auch $\xrightarrow[x]{x}$ betrachtet werden, anstelle von $\xrightarrow{\varepsilon}$ auch $\xrightarrow[\varepsilon]{\varepsilon}$ und von $\xrightarrow{?}$ auch $\xrightarrow[?]{?}$.

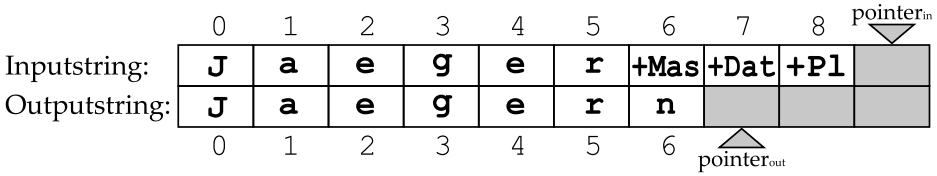

Abbildung 4: Status des Eingabe- und Ausgabestrings nach dem erstmaligen Erreichen eines terminalen Knotens.

Beim Erreichen eines terminalen Symbols wird überprüft, ob $pointer_{in}$ auf kein weiteres Eingabesymbol zeigt. Ist dies der Fall, wird der Ausgabestring von Position 0 bis $pointer_{out} - 1$ in die Ergebnisliste geschrieben. Damit ist `J, a, e, g, e, r, n` eine gültige Interpretation von `J, a, e, g, e, r, +Mas, +Dat, +Pl`.

Es wird mit der Bearbeitung des ersten Stapelspeicher-Eintrags fortgefahren: $6 \xrightarrow[s]{+Mas} 8$ bei $pointer_{in} = 6$ und $pointer_{out} = 6$. Der Übergang wird traversiert, das Symbol `s` an Position 6 in den Ausgabestring geschrieben und beide Zeiger inkrementiert. da ab Knoten 8 jedoch kein Übergang existiert, der das Symbol `+Dat` akzeptiert, wird die Tiefensuche abgebrochen und der nächste Eintrag vom Stapelspeicher genommen: $6 \xrightarrow[\varepsilon]{+Mas} 9$ bei $pointer_{in} = 6$ und $pointer_{out} = 6$. Von dort kann der Übergang $9 \xrightarrow[\varepsilon]{+Dat} 11$ traversiert werden, doch von Zustand 11 geht keine Kante ab, die das Folgesymbol `+Pl` akzeptiert. Die Tiefensuche wird wiederholt abgebrochen und da der Stapelspeicher keine weiteren Einträge enthält, ist die Übersetzung des Eingabestrings beendet. Die Ausgabemenge enthält einen Eintrag: `J,a,e,g,e,r,n`.

4 Implementation

Während die abstrakte Übersetzung eines Eingabestrings in einen Ausgabestring mittels eines Transducers simpel erscheint, gibt es für die maschinelle Implementation des Algorithmus mehrere Möglichkeiten, gerade in Bezug auf die Verarbeitung der Übergangsfunktion δ.

Im Folgenden werden zwei unterschiedliche Implementationen, die im Rahmen dieser Arbeit erstellt wurden, vorgestellt und miteinander verglichen. Als Basis dienen die beiden gängigsten Repräsentationsvarianten für Automaten: Die Matrix- und die Listenrepräsentation, wobei beide Varianten unter Berücksichtigung der speziellen Vor- und Nachteile der Programmiersprache Perl implementiert wurden.

Um die Implementationen bewerten zu können, wurden Geschwindigkeitstests

(*Benchmark-Tests*) durchgeführt. Die Test-Konstellation wird im Folgenden kurz erläutert, bevor die Implementationsvarianten vorgestellt werden.

4.1 Benchmark-Tests

Für die Verwendung des Interpreters im Morphologie-Modul des Syntax-Parsers wurde mittels *lexc* und *xfst* ein umfangreiches Lexikon erstellt (s. Tab. 1). Es enthält starke und schwache Substantive und Verben, Adjektive (auch unregelmäßig zu steigernde), bestimmte und unbestimmte Artikel, Personal-, Possessiv- und Demonstrativpronomen, Präpositionen und Konjunktionen.[16] Keine Berücksichtigung fanden hierbei komplexere morphologische Wortbildungsprozesse wie Komposition und Derivation. Die entsprechenden Wortformen wurden direkt im Lexikon eingetragen, jedoch im Fall einiger Affixe gesondert realisiert. Etwa wurden für Suffixe wie „-heit“ oder „-keit“ und Präfixe wie „un-“ in *lexc* Fortsetzungslexika angelegt.

Zustände	$\lvert\Sigma_1 \cup \Sigma_2\rvert$	**Kanten**	**Pfade**	**Analyseformen**	**Oberflächenformen**
4611	88	9036	106717	105191	51199

Tabelle 1: Lexikon-Umfang

Dieses Lexikon dient auch als Basis für die Implementationsgeschwindigkeitsmessung. Da die Geschwindigkeit der Übersetzung eines Eingabestrings von seiner Länge und der Größe des Transducers abhängt, werden pro Implementation Benchmark-Tests über drei verschiedene Texte durchgeführt. Die untersuchten Texte sind frei verfügbar im Rahmen des internationalen Gutenberg-Projekts.

Fontane: Theodor Fontane – Effi Briest (1895)[17]

Carroll: Lewis Carroll – Alice im Wunderland (1865)[18]

Kafka: Franz Kafka – Die Verwandlung (1915)[19]

Für die Vorverarbeitung wurden die Texte tokenisiert und in eine Wortliste übertragen, die nur Zeichenfolgen enthält, die ausschließlich aus Buchstaben bestehen,

[16] Als Nachschlagewerke für die Erstellung des Lexikons dienten in erster Linie Brückner und Sauter (1986a,b); Wellmann (1975); Kühnhold et al. (1978).

[17] `http://www.gutenberg.org/etext/5323`; zuletzt abgerufen: 10.11.2008.

[18] Aus dem Englischen von Antonie Zimmermann. `http://www.gutenberg.org/etext/19778`; zuletzt abgerufen: 10.11.2008.

[19] `http://www.gutenberg.org/etext/22367`; zuletzt abgerufen: 10.11.2008.

	Fontane	**Carroll**	**Kafka**
Wörter im Text (Token):	97451	28271	22046
Wörter im Text (Types):	11819	4573	4423
∅ **Wortlänge**:	≈ 4,9996	≈ 4,90602	≈ 5,23832
Übersetzte Wörter (Token):	≈ 55,97%	≈ 53,35%	≈ 48,25%
Übersetzte Wörter (Types):	≈ 16,36%	≈ 22,26%	≈ 21,21%
∅ **Interpretationen**:	≈ 3,24667	≈ 3,5548	≈ 3,90589
Nicht übersetzte Wörter (Token):	≈ 44,03%	≈ 46,65%	≈ 51,75%
Nicht übersetzte Wörter (Types):	≈ 83,64%	≈ 77,74%	≈ 78,79%

Tabelle 2: Abdeckung des Lexikons in Bezug auf die drei Benchmark-Texte.

die Teil des Lexikon-Alphabets sind.[20] Umlaute und das „ß“ werden umschrieben in „ae“, „oe“, „ue“ und „ss“, Großbuchstaben werden in Kleinbuchstaben überführt. Da die Abdeckung des Lexikons verschieden ist, werden die Zeiten für erfolgreich übersetzte Wörter und nicht erfolgreich übersetzte Wörter getrennt angegeben.[21]

Wie die ≈ 50%ige Token- im Vergleich zur ≈ 20%igen Typeabdeckung in Tabelle 2 zeigt, wurde das Lexikon besonders unter Berücksichtigung hochfrequenter Wortformen erstellt. Wird ein Token gefunden, werden durchschnittlich 3 bis 4 Interpretationen zurückgegeben, was einen erheblichen Aufwand für die Disambiguierung im Syntaxmodul bedeutet, da die Interpretationen in der Ausgabe nicht gewichtet werden.

4.2 Matrixbasierte Variante

Matrizen können der Beschreibung eines Transducers dienen, indem sie den nächsten zu erreichenden Zustand als Funktion des aktuellen Zustands und der von ihm abgehenden Kanten wiedergeben.

Tabelle 3 stellt das „Jaeger“-Lexikon (s. S. 13) als Matrix dar. Die erste Spalte führt alle Knoten des Netzes auf, in ihren Reihen sind zu den jeweiligen Kantenbeschreibungen (dargestellt in der ersten Reihe) die Zielknoten aufgeführt. Die Label werden als Paar $label_{up}$:$label_{down}$ definiert.

Die Traversierung geschieht durch Nachschlagen der Zielknoten in den Spalten des nächsten Inputsymbols und der Reihe des aktuellen Zustands. Diese Form der

[20] Hierbei wird davon ausgegangen, dass $\Sigma_1 = \Sigma_2$.

[21] Die Tests wurden durchgeführt auf einem Intel Pentium 4 (3.20GHz (1.6GHz effektiv genutzt), 2GB RAM) unter Windows XP SP3 und ActiveState perl 5.8.8.

	J J	a a	e e	g g	r r	+Mas s	+Mas ε	+Mas n	+Gen ε	+Dat ε	+Acc ε	+Nom ε	+Sg ε	+Pl ε
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	2	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	3	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	4	0	0	0	0	0	0	0	0	0	0
4	0	0	5	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	6	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	8	9	7	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	10	0	0	0	0
8	0	0	0	0	0	0	0	0	11	0	0	0	0	0
9	0	0	0	0	0	0	0	0	10	11	12	12	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	13
11	0	0	0	0	0	0	0	0	0	0	0	0	13	0
12	0	0	0	0	0	0	0	0	0	0	0	0	13	13
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabelle 3: Darstellung des „Jaeger"-Transducers als Matrix (Notation nach Koskenniemi, 1983). $S = 0$, $F = \{13\}$.

Traversierung ist sehr effizient, da es sich um einfache Zugriffe auf Arrays handelt.

Die vorliegende Implementation in Perl (s. Anhang A.2) speichert die Kantenbeschreibungen redundant in Hashes um je nach Übersetzungsrichtung schnell auf die als Referenz gespeicherten Listen mit den Zielknoten zugreifen zu können.

Matrixrepräsentationen sind in ihrer ursprünglichen Form nur für die Beschreibung von deterministischen Automaten geeignet. Da das *XEROX Finite State Tool* deterministische FSTs erzeugt, fällt diese Einschränkung nicht ins Gewicht.

Benchmark

Alle Wörter der Eingabetexte wurden nacheinander übersetzt und sowohl durchschnittliche, minimale und maximale Übersetzungszeiten ermittelt.

Wie in Tabelle 4 zu sehen, liegt die durchschnittliche Übersetzungszeit eines im Transducer vorhandenen Wortes bei etwa 0,00366 Sekunden, für durchschnittlich 3 bis 4 Interpretationen je Wort. Maximal liegt die Zeit bei $\approx$ 0,025 Sekunden, minimal bei $\approx$ 0,00015 Sekunden. Die durchschnittliche Zeit verkürzt sich um circa 0,00153 Sekunden, wenn ein Wort nicht im Lexikon gefunden wurde und entsprechend die Verarbeitung vorzeitig abgebrochen werden kann. Die minimalen und maximalen Übersetzungszeiten schwanken in Bezug auf die übersetzten bzw. nicht übersetzten

	Fontane	**Carroll**	**Kafka**
Übersetzte Wörter:			
∅ **Interpretationszeit** (s):	≈ 0,0036	≈ 0,0035	≈ 0,00387
Min Interpretationszeit (s):	≈ 0,00012	≈ 0,00015	≈ 0,00017
Max Interpretationszeit (s):	≈ 0,02679	≈ 0,0232	≈ 0,02492
Nicht übersetzte Wörter:			
∅ **Interpretationszeit** (s):	≈ 0,00222	≈ 0,00203	≈ 0,00214
Min Interpretationszeit (s):	≈ 0,0001	≈ 0,00027	≈ 0,00017
Max Interpretationszeit (s):	≈ 0,02309	≈ 0,01601	≈ 0,01772

Tabelle 4: Benchmark für matrixbasierten Transducer.

Wörter.

4.3 Listenbasierte Variante

Eine alternative Möglichkeit bietet sich durch die Abbildung des Netzes mittels Listen an. Hierbei werden von einem Zustand ausgehend alle abgehenden Kanten mit ihren Labeln und ihren Zielknoten in einer Liste gespeichert. Die Übersetzung eines Eingabestrings beginnt hierbei am Startknoten S und überprüft alle abgehenden Kanten auf ihre Traversierbarkeit. Im Falle einer Traversierung wird der zugehörige Zielknoten nachgeschlagen und von dort aus weiter übersetzt.

Die vorliegende Implementation (s. Anhang A.3) mischt hierfür die Verwendung von Hashes und Arrays um eine möglichst performante und speichereffiziente Repräsentation des Lexikons zu erreichen. Terminale Übergänge werden anstelle von terminalen Knoten eingesetzt (ein Zustand ist dadurch terminal, dass er eine ausgehende terminale Kante besitzt).

Die Listen-Speicherung hat gegenüber der Matrix-Speicherung zwei Vorteile: Sie lässt sich auch bei nicht-deterministischen Automaten einsetzen und ist auch bei „lichten“ Graphen, d.h. Graphen mit vielen Knoten aber einer geringen Vernetzung dieser Knoten untereinander, einsetzbar.[22]

	Fontane	**Carroll**	**Kafka**
Übersetzte Wörter:			
∅ **Interpretationszeit** (s):	≈ 0,0027	≈ 0,00289	≈ 0,0032
Min Interpretationszeit (s):	≈ 0,00013	≈ 0,00015	≈ 0,00013
Max Interpretationszeit (s):	≈ 0,026	≈ 0,02748	≈ 0,02588
Nicht übersetzte Wörter:			
∅ **Interpretationszeit** (s):	≈ 0,00069	≈ 0,00067	≈ 0,0007
Min Interpretationszeit (s):	≈ 0,00011	≈ 0,00019	≈ 0,00012
Max Interpretationszeit (s):	≈ 0,01574	≈ 0,01586	≈ 0.01563

Tabelle 5: Benchmark für listenbasierten Transducer.

Benchmark

Die durchschnittliche Interpretationszeit pro gefundenem Token im listenbasierten Transducer (s. Tab. 5) liegt bei etwa 0,00293 Sekunden und damit etwa 4,26 Mal höher als die Bearbeitungszeit für nicht im Lexikon enthaltene Wörter. Das heißt, falls ein Wort nicht im Lexikon enthalten ist, kann schnell abgebrochen werden, während bei im Lexikon enthaltenen Wörtern erwartbar der Bearbeitungsstack bei durchschnittlich 3 bis 4 Interpretationen pro Token erheblich mehr Berechnungszeit erfordert.

Die minimalen Interpretationszeiten liegen sowohl bei gefundenen als auch bei nichtgefundenen Wörtern mit ca. 0,00014 etwa gleichauf. Dies lässt auf eine geringe Anzahl alternativer Kantenübergänge während der Interpretation schließen und bei nicht gefundenen Wörtern auf einen schnellen Abbruch aus demselben Grund. Die maximale Bearbeitungszeit entspricht bei gefundenen Wörtern etwa dem 1,68-fachen des Wertes bei nicht gefundenen Wörtern. In beiden Fällen kann wieder von einer hohen Dichte alternativer Kantenübergänge ausgegangen werden, die jedoch bei nicht gefundenen Wörtern nicht zur vollständigen Pfadtraversierung führen.

4.4 Bewertung

Die durchchnittliche Übersetzungszeit für sowohl gefundene als auch nicht gefundene Wörter im Lexikon ist im Fall des listenbasierten Transducers (≈ 0.00293 Sek. / ≈ 0.00069 Sek.) deutlich niedriger im Vergleich zum matrixbasierten Transducer (≈ 0.00366 Sek. / ≈ 0.00213 Sek.). Dies dürfte in erster Linie an der Art des Lexikons lie-

[22] Siehe hierzu die hohe Dichte unbesetzter Zellen in Tabelle 3, die zu einem hohen Speicheraufwand führen kann.

gen, das ein sehr hohes Vorkommen von ε-Kanten aufweist. Während bei der Listenverarbeitung lediglich die von dem derzeit aktuellen Zustand ausgehenden Kanten untersucht werden müssen, ist im Fall der Matriximplementation für jeden Zustand nachzusehen, ob eine ε-Kante von ihm ausgeht, wofür zunächst alle ε-Listen konsultiert werden müssen.

5 Zusammenfassung

Diese Hausarbeit behandelte die Erstellung eines morphologischen Lexikons mit Hilfe des *XEROX Finite State Tools* und dessen Interpretation in zwei Implementationsvarianten, umgesetzt in der Programmiersprache Perl. Ziel war die Erstellung eines Morphologie-Moduls zum Einsatz in einer Syntax-Parsing-Umgebung.

Im Fall des Lexikons wurde eine durchschnittliche Abdeckung von $\approx$ 52,52% für die drei Beispieltexte erreicht, was zwar weit von einem möglichen Einsatz in einer produktiven Analyseumgebung ist, mir jedoch für Demonstrationszwecke eine befriedigende Größe zu sein scheint. Ebenfalls für diese Zwecke befriedigend ist die Interpretationszeit von ca. 0,00289 Sekunden pro Wort im Fall des matrixbasierten Transducers[23] und ca. 0,00121 Sekunden im Fall des listenbasierten Transducers.

Der listenbasierte Transducer erwies sich im Vergleich zum matrixbasierten Transducer als performanter, wobei der größte Nachteil der Matrixvariante in der ständigen Konsultation der ε-Kanten-Listen zu verorten ist. Eine effizientere Implementation dieser „besonderen“ Listen könnte die Performanz dieser Variante deutlich erhöhen, was jedoch nicht Gegenstand dieser Hausarbeit war.

Aufgrund der flexibleren Gestaltung und der höheren Geschwindigkeit, kommt in der Syntax-Parsing-Applikation die listenbasierte Variante des Interpreters zum Einsatz. Sie wurde um zahlreiche Funktionen erweitert, die nicht primär für die Übersetzung eines Lexikons notwendig sind, etwa der Unterstützung des „?“-Symbols des *XEROX Finite State Tools* oder die interne Protokollierung von ε-Schleifen, um endlose Verarbeitungen und unendlich große Ergebnismengen zu verhindern.

Das Projekt wurde in Zusammenarbeit mit Daniel Jettka durchgeführt, der basierend auf der Ausgabe des Transducers syntaktische Analysen in XSLT durchführt.

[23] Gemittelte Interpretationszeit übersetzter und nicht übersetzter Wörter ohne Berücksichtigung der Abdeckung.

A Programm-Dokumentation

Das *XEROX Finite State Tool* bietet die Möglichkeit an, bereits kompilierte Automaten in Form von Kantenübergangsbeschreibungen als Prolog-Fakten zu exportieren.[24] In dieser Form lässt sich das Netz leicht in selbstgeschriebene Programme einlesen, ohne eine erneute Kompilation zu erfordern. Die hier vorgestellten Interpreter erfüllen dabei die selbe Aufgabe wie das *XEROX*-Programm *Lookup*: Es überführt eingegebene Strings in eine Menge von Analysestrings oder einen Analysestring in eine Menge möglicher Oberflächenformen.

Um nicht bei jedem Aufruf von Neuem den Transducer auf Basis der Prologdatei aufbauen zu müssen, wird mithilfe des Storable-Moduls[25] das generierte Lexikon-Objekt als persistente Datenstruktur in eine Datei geschrieben, die bei neuerlichem Aufruf anstelle der Prolog-Datei geladen wird. Für die Verwendung des Transducers in der Online-Demonstration muss daher bezüglich der Verarbeitungsgeschwindigkeit neben der Übersetzungszeit auch die Netzerstellung berücksichtigt werden. Diese kann bei einem Parsen der Prolog-Datei $\approx$ 0,44 Sekunden dauern, beim Laden der serialisierten Datei $\approx$ 0,092 Sekunden.[26]

A.1 Ausgabe nach den TEI Guidelines

Jede Interpretation eines eingegebenen Strings wird als Menge „flacher“ Symbol-Listen zurückgegeben. Dabei sind Einzelsymbole Teil des Stammes, Mehrzeichen-Symbole repräsentieren die morphologischen Informationen.

Um diese Informationen in eine für das Syntax-Modul interpretierbare Form zu überführen, werden alle Mehrsymbol-Zeichen durch einen zweiten Transducer in Attribut-Wert-Paare übersetzt. Aus `J, a, e, g, e, r, +Mas, +Dat, +Pl` wird dadurch `J, a, e, g, e, r, (GEN, mask), (KAS, dat), (NUM, pl)`.

Diese Informationen werden durch DOM-Operationen auf ein XML::LibXML-Objekt[27] in eine XML-Instanz überführt, die den TEI-Richtlinien für Merkmals-Lexika entspricht.[28]

[24] Der entsprechende Befehl lautet „write prolog“.

[25] `http://search.cpan.org/~ams/Storable-2.18/Storable.pm`; zuletzt abgerufen am 15.11.2008.

[26] Dies bezieht sich auf die Systemangaben in Fußnote 21. Bei der Online-Demonstration fallen zusätzliche Belastungen durch die Verwendung des Servers für weitere Web-Angebote ins Gewicht, was sowohl die Geschwindigkeit der Netzgenerierung als auch der Übersetzung negativ beeinflusst.

[27] `http://search.cpan.org/~pajas/XML-LibXML-1.68/`; zuletzt abgerufen am 10.11.2008.

[28] Burnard und Bauman (2008). Die genauen Merkmalsbeschreibungen und ihre Struktur wurden

A.2 Modul: MatrixFST

ZUSAMMENFASSUNG

```
use MatrixFST;

use constant {
  DOWN => 1,
  UP => 2,
};

my $x = MatrixFST->new('lexicon.plg');
my @interp = $x->parse(DOWN,'pferdes');
print join("\n", map(join('', @$_), @interp));
```

BESCHREIBUNG

Definition eines Finite State Transducers mit Parsing-Routinen über eine Matrix.

VERWENDUNG

new(file)

Der Konstruktor. Gibt ein MatrixFST-Objekt zurück. Benötigt eine Prolog-Lexikon-Datei.

```
my $fst = MatrixFST->new(file);
```

readPrologFile(file)

Liest eine Prolog-Faktendatei ein, wie sie von Xerox FST generiert wird. Anhand der Kanten- und Sigma-Informationen wird ein FST erstellt.

parse(way, inputarray)

Eine Inputzeichenkette (als Referenz auf ein Array) wird in die Angegebene Richtung übersetzt. Zurückgegeben wird eine Referenz auf ein Array der Analysestrings.

```
$fst->parse(1,['a','b','c']);
```

den Erfordernissen des Syntax-Moduls entsprechend von Daniel Jettka angepasst.

A.3 Modul: FST

ZUSAMMENFASSUNG

```
use FST;

my $fst = FST->new();
$fst->edge(0,1,'a');
$fst->edge(1,2,'b','c');
$fst->edge(1,2,'b','d');
$fst->edge(2,3,0,'d');
$fst->end(2);
$fst->end(3);
my $interp = $fst->parse(1,['a','b']);
print join(", ", map(join('', @$_), @$interp));
# Gibt zurück: ac, acd, ad, add
```

BESCHREIBUNG

Definition eines Finite State Transducers mit Parsing-Routinen.

VERWENDUNG

new()

Der Konstruktor. Gibt ein *FST*-Objekt zurück. Dem Konstruktor können verschiedene Parameter übergeben werden.

```
my $fst = FST->new(
  # Aktivieren des Trace-Modus.
  # Default ist 0 (aus).
    'trace'      => 1,
  # Maximale Größe des Zeigerstacks.
  # Default ist 1000.
    'pointermax' => 100,
  # Maximale Größe des Interpretationstacks.
  # Default ist 256.
    'stackmax'   => 50,
  # Maximale Anzahl möglicher
```

```
    # Epsilonschleifentraversierungen.
    # Default ist 1.
      'epsilonmax' => 5
  );
```

edge(start, target, label1, [label2])

Fügt eine neue Kante dem Netz zu. Erster Parameter ist der eindeutige Bezeichner des Startknotens, zweiter Parameter ist der eindeutige Bezeichner des Endknotens, dritter Parameter ist die Kantenbezeichnung. Bei übersetzenden Kanten kann optional ein vierter Parameter angegeben werden, der als untere Kantenbezeichnung dient. In diesem Fall ist der dritte Parameter die obere Kantenbezeichnung. Kantenbezeichnungen werden in Σ aufgenommen.

```
$fst->edge('node1', 'node2', 'b', 'd');
```

sigma(@symbol)

Fügt die Elemente einer übergebenen Symbolmenge in Σ ein.

```
$fst->sigma('a','b','c');
```

end(node)

Der Knoten des übergebenen Knotenbezeichners wird als terminaler Knoten gesetzt.

```
$fst->end('node5');
```

parse(way, string)

Eine Inputzeichenkette (als Referenz auf ein Array) wird in die angegebene Richtung übersetzt. Zurückgegeben wird eine Referenz auf ein Array der Analysestrings.

```
$fst->parse(1,['a','b','c']);
```

A.4 Modul: XEROX

VORAUSSETZUNG

```
use FST;
```

ZUSAMMENFASSUNG

```
use XEROX;
my $x = XEROX->new();
$x->readPrologFile('lexicon.plg');
my $interp = $t->parseTokenDown('pferdes');
print join("\n", map(join('', @$_), @$interp));
```

BESCHREIBUNG

Die Klasse *XEROX* erbt von der Basisklasse *FST* die Grundfunktionen zum Aufbau eines Finite State Transducers und fügt Methoden zum Umgang mit Übersetzern des Xerox FST Toolkits hinzu.

VERWENDUNG

new()

Der Konstruktor. Gibt ein *XEROX*-Objekt zurück. Für die verschiedenen Parameter s. die Basisklasse *FST*.

readPrologFile(filename)

Liest eine Prologfakten-Datei ein, wie sie von Xerox FST generiert wird. Anhand der Kanten- und Σ-Informationen wird ein FST erstellt.

getSigma()

Gibt die Menge Σ zurück.

getMulticharSymbols()

Gibt die Menge aller Mehrzeichensymbole zurück.

parseTokenUp(string)

Übersetzt eine gegebene Zeichenkette in die obere Sprache. Gibt einen Analysestring zurück.

parseTokenDown(string)

Übersetzt eine gegebene Zeichenkette in die untere Sprache. Gibt einen Analysestring zurück.

A.5 Modul: Interpretation

ZUSAMMENFASSUNG

```
use Interpretation;
use XEROX;

my $lex = XEROX->new();
$lex->readPrologFile('lexicon.plg');

my $feat = XEROX->new();
$feat->readPrologFile('features.plg');

my $token = 'pferdes';

my $inter = Interpretation->new(
              $token,
              $feat,
              $lex->parseTokenUp($token),
              'up'
            );
```

BESCHREIBUNG

Die Klasse Interpretation gibvt ein Objekt zurück, das die analysierten morphologischen Eigenschaften zu einem Token speichert.

VERWENDUNG

new(string, lexicon, \@string, 'up'|'down')

Der Konstruktor. Gibt ein *Interpretation*-Objekt zurück. Dem Konstruktor muss ein Token, ein Feature-Lexikon, eine Liste mit Analysestrings und die Richtung der Analyse übergeben werden.

```
my $inter = Interpretation->new(
              $token,
              $feat,
              $lex->parseTokenUp($token),
              'up'
            );
```

`getCount()`

Gibt die Anzahl aller Interpretationen zurück.

`getInterpretations()`

Gibt die Liste aller Interpretationen zurück.

A.6 Modul: Morpho

VORAUSSETZUNG

```
use XEROX;
use Interpretation;
use Storable;
```

ZUSAMMENFASSUNG

```
use Morpho;

my $net = Morpho->new(
  'net'   => 'Data/mylexicon.plg',
  'feat'  => 'Data/features.plg',
  'store' => 'Data/mylexicon.net'
);

my @interp = $net->parseSentenceUp('Dies ist ein Beispielsatz.');
```

BESCHREIBUNG

Die Klasse *Morpho* stellt die Schnittstelle zwischen den Klassen *XEROX* und der Klasse *Interpretation* dar. Es bietet eine Parsing-Schnittstelle für Token und Sätze an und gibt die Interpretationen als *Interpretation*-Objekte zurück.

VERWENDUNG

new()

Der Konstruktor. Gibt ein *Morpho*-Objekt zurück. Übergeben werden können Dateinamen für Prologfakten-Lexika, wie sie XFST generiert. Zum einen für das morphologische Lexikon ('net'), zum anderen für die Übersetzung der Mehrzeichensymbole in Attribut-Wert-Paare ('feat'). Optional gibt es die Möglichkeit, mit einer vorkompilierten Version dieser Dateien zu arbeiten ('store').[29]

[29] Dabei handelt es sich um eine serialisierte Perl-Datenstruktur. Verwendet wird hierfür das CPAN-Modul *Storable* (`http://search.cpan.org/~ams/Storable-2.18/Storable.pm`; zuletzt abgerufen am 8.10.08).

getSigma()

Gibt die Menge Σ zurück.

getMulticharSymbols()

Gibt die Menge aller Mehrzeichensymbole zurück.

parseTokenUp(string)

Übersetzt ein übergebenes Token in die obere Sprache. Gibt ein *Interpretation*-Objekt zurück.

parseTokenDown(string)

Übersetzt ein übergebenes Token in die untere Sprache. Gibt ein *Interpretation*-Objekt zurück.

parseSentenceUp(string)

Übersetzt einen übergebenen Satz in die obere Sprache. Gibt ein Array mit *Interpretation*-Objekten zurück.

parseSentenceDown(string)

Übersetzt einen übergebenen Satz in die untere Sprache. Gibt ein Array mit *Interpretation*-Objekten zurück.

A.7 Modul: Interpretation::Result

VORAUSSETZUNG

```
use XML::LibXML;
```

ZUSAMMENFASSUNG

```
use Morpho;
use Interpretation::Result;

my $net = Morpho->new(
  'net'   => 'Data/mylexicon.plg',
  'feat'  => 'Data/features.plg'
);

my $result = Interpretation::Result->new();

foreach my $interp ($net->parseSentenceDown('Dies ist ein Beispiel-Satz.')) {
  $result->addInterpretation($interp);
};

print $result->toTEI();
```

BESCHREIBUNG

Erstellt ein XML-Dokument[30] entsprechend einer TEI-konformen Grammatik zur Modellierung von Lexikon-Informationen.

VERWENDUNG

new()

Der Konstruktor. Gibt ein *Interpretation::Result*-Objekt zurück.

getUnknown()

[30] Hierzu wird das CPAN-Modul *XML::LibXML* verwendet (`http://search.cpan.org/~pajas/XML-LibXML-1.66/`; zuletzt abgerufen am 8.10.08).

Gibt eine Liste aller geparsten Tokens zurück, die keinen Eintrag im Lexikon besitzen.

complete()

Gibt einen wahren Wert zurück, wenn alle geparsten Token im Lexikon gefunden werden konnten, andernfalls einen unwahren Wert.

addInterpretation(object)

Fügt einen Eintrag in Form eines *Interpretation*-Objekts dem Lexikon hinzu.

toTEI()

Gibt das Lexikon als TEI-konformes XML-Dokument zurück.

toHTML()

Gibt das Lexikon als HTML-Liste zurück.

toText()

Gibt das Lexikon als formatierten Text zurück.

B Beispiel-Lexikon

```
1  Multichar_Symbols
2  +1P +2P +3P                ! Person
3  +Sg +Pl                    ! Numerus
4  +Pres +Imp +Perf +Temp     ! Tempus
5  +MInd +MImp +MKonj         ! Modus
6  +Verb +Mark1 +Mark2
7  +Pref +NoPref
8
9  Definitions
10 !Präfixe: be, er, ver, zer, emp, ent, ge
11   be  = {be} :["+NoPref" {BE} ];
12   ver = {ver}:["+NoPref" {VER}];
13
14 LEXICON Root
15   0:+Pref    Verbs;
16
17 LEXICON Verbs
18   < [0|ver|be] {lach}>    VerbMarker1;
19   < [0|ver|be] {helf}>    VerbMarker2;
20
21 LEXICON VerbMarker1
22   +Verb:+Verb+Mark1 VerbEnd;
23
24 LEXICON VerbMarker2
25   +Verb:+Verb+Mark2 VerbEnd;
26
27 LEXICON VerbEnd
28 +1P+Sg+Pres+MInd:e+1P+Sg+Pres+MInd   #;
29 +2P+Sg+Pres+MInd:st+2P+Sg+Pres+MInd  #;
30 +3P+Sg+Pres+MInd:t+3P+Sg+Pres+MInd   #;
31 +1P+Pl+Pres+MInd:en+1P+Pl+Pres+MInd  #;
32 +2P+Pl+Pres+MInd:t+2P+Pl+Pres+MInd   #;
33 +3P+Pl+Pres+MInd:en+3P+Pl+Pres+MInd  #;
34
35 +1P+Sg+Imp+MInd:+Temp+1P+Sg+Imp+MInd     #;
36 +2P+Sg+Imp+MInd:+Tempst+2P+Sg+Imp+MInd  #;
37 +3P+Sg+Imp+MInd:+Temp+3P+Sg+Imp+MInd     #;
38 +1P+Pl+Imp+MInd:+Tempen+1P+Pl+Imp+MInd  #;
39 +2P+Pl+Imp+MInd:+Tempt+2P+Pl+Imp+MInd   #;
40 +3P+Pl+Imp+MInd:+Tempen+3P+Pl+Imp+MInd  #;
```

```
41
42 < 0:t ["+1P"|"+2P"|"+3P"] ["+Sg"|"+Pl"] "+Perf" > #;
43
44 < 0:[0|e] "+Sg" "+MImp" > #;
45
46 < 0:t "+Pl" "+MImp" > #;
```

verb.lexc (komplett)

```
1  read lexc < verb.lexc
2  def lexVerb;
3
4  def 2P3PSgPres [["+2P"|"+3P"] ?* "+Sg" ?* "+Pres"];
5
6  # Präsens/Imperativ:
7  def PresM2a [e -> I || _ ?* "+Mark2" ?* [2P3PSg|"+Sg" "+MImp"] ?*];
8  def PresM2b [e -> 0 || ?* I ?* "+Mark2" ?* _ "+Sg" "+MImp" ?*];
9  def PresM2 PresM2a .o. PresM2b .o. [I -> i];
10
11 # Imperfekt:
12 def ImpM1 [["+Temp" (e)] -> {te} ||
13            ?* "+Mark1" ?* _ \e ?* "+Imp" ?*];
14 def ImpM2 [e -> a || _ ?* "+Mark2" ?* "+Imp" ?*];
15 def Imp ImpM1 .o. ImpM2;
16
17 # Perfekt
18 def PerfEnd [ t -> {en} || "+Mark2" ?* _ ?* "+Perf" ?*];
19 def PerfM2 [e -> o || _ ?* "+Mark2" ?* "+Perf" ?*];
20 def Perf PerfEnd .o. PerfM2;
21
22 def gePrefix ["+Pref" -> {ge} ||
23               .#._ [? - "+NoPref"] ?* "+Perf" ?*];
24
25 def specialChars [E -> e, B -> b, V -> v, R -> r];
26
27 def Eraser [["+1P"|"+2P"|"+3P"|
28              "+Sg"|"+Pl"|
29              "+Pres"|"+Imp"|"+Perf"|"+Temp"|
30              "+Mark1"|"+Mark2"|
31              "+MInd"|"+MImp"|"+MKonj"|
32              "+Pref"|"+NoPref"|
33              "+Verb"] -> 0];
34
```

```
35 clear stack;
36 read regex lexVerb .o. PresM2 .o. Imp .o. Perf
37       .o. gePrefix .o. specialChars .o. Eraser;
```

Literatur

Brückner, Tobias und Christa Sauter (Hrsg.) (1986a): *Band I: Judäa – Saman*. Rückläufige Wortliste zum heutigen Deutsch. Institut für deutsche Sprache, Mannheim, Zweite Auflage.

Brückner, Tobias und Christa Sauter (Hrsg.) (1986b): *Band II: Ataman – Jazz*. Rückläufige Wortliste zum heutigen Deutsch. Institut für deutsche Sprache, Mannheim, Zweite Auflage.

Burnard, Lou und Syd Bauman (Hrsg.) (2008): *TEI P5: Guidelines for Electronic Text Encoding and Interchange*, Kap. 18: Feature Structures. TEI Consortium, Erste Auflage, S. 541–575. Standardisiert als ISO 24610-1:2006. Language resource management – Feature structures – Part 1: Feature structure representation.
URL `http://www.tei-c.org/release/doc/tei-p5-doc/en/Guidelines.pdf`

Fitschen, Arne (2004): *Ein Computerlinguistisches Lexikon als komplexes System*. Dissertation, Universität Stuttgart.

Jettka, Daniel (2008): *XSLT-basiertes Syntax-Parsing unter Verwendung einer Unifikationsgrammatik*. Hausarbeit.

Karttunen, Lauri und Kenneth R. Beesley (1992): *Two-level rule compiler. ISTL-92-2. Forschungsbericht*, Xerox Palo Alto Research Center, Palo Alto, CA.
URL `http://www.xrce.xerox.com/competencies/content-analysis/fssoft/docs/twolc-92/twolc92.html`

Karttunen, Lauri und Kenneth R. Beesley (2001): *A short history of two-level morphology*. In: *13th European Summer School in Logic, Language and Information*. ESSLLI 2001.
URL `http://www2.parc.com/istl/members/karttune/publications/esslli-2001/twol-history.pdf`

Koskenniemi, Kimmo (1983): *Two-Level Model for Morphological Analysis*. In: *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Ausgabe 2. International Joint Conference on Artificial Intelligence, Inc., S. 683–685.

Kühnhold, Ingeburg, Oskar Putzer und Hans Wellmann (1978): *Deutsche Wortbildung – Typen und Tendenzen in der Gegenwartssprache. Dritter Hauptteil: Das Substantiv*, Ausgabe 43 von *Schriften des Instituts für deutsche Sprache Mannheim*. Pädagogischer Verlag Schwann, Düsseldorf.

Porter, Martin F. (1980): *An algorithm for suffix stripping*. In: *Program*, 14, 3, S. 130–137.
URL `http://tartarus.org/~martin/PorterStemmer/def.txt`

Wellmann, Hans (1975): *Deutsche Wortbildung – Typen und Tendenzen in der Gegenwartssprache. Zweiter Hauptteil: Das Substantiv*, Ausgabe 32 von *Schriften des Instituts für deutsche Sprache Mannheim*. Pädagogischer Verlag Schwann, Düsseldorf.