Ruhr-Universität Bochum
Sprachwissenschaftliches Institut

Dozent: Prof. Dr. Tibor Kiss
Hauptseminar: *„Syntaktische Verarbeitung I: Tagging und Korpora“*
Sommersemester 2005

„Hinweise zur Installation und Verwendung des Constraint-Taggers“

vorgelegt von:

Alex Linke
Lohackerstr. 63
44867 Bochum
Alexander.Linke-2@rub.de
Linguistik/Philosophie, B.A.
4. Fachsemester

Rona Linke
Lohackerstr. 63
44867 Bochum
Rona.Linke@rub.de
Linguistik/Slavistik, B.A.
4. Fachsemester

Björn Wilmsmann
Stefan-George-Str. 15a
46117 Oberhausen
bjoern@uni-bottrop.de
Linguistik/Anglistik, B.A.
4. Fachsemester

Inhaltsverzeichnis

1 Installation der Software

1.1 Installation der Abhängigkeiten

1.1.1 Lingua::Tokenize

Das Perl-Modul *Lingua::Tokenize* ist im Rahmen des Gesamtablaufs der zum Tagging notwendigen Arbeitsschritte sowohl für die eigentliche Tokenisierung, als auch für das Normalisieren der Satzanfänge auf Grund statistischer Abwägung (*Korpus-Filter*) zuständig. Auch eine Segmentierung von Input zu Sätzen fällt in den Funktionsumfang dieses Moduls.

Eine detaillierte Dokumentation des Moduls liegt im Perl-eigenen POD Format vor und lässt sich durch den Aufruf des Kommandos `perldoc Lingua::Tokenize` in einem Terminal, bzw. der Eingabeaufforderung von Microsoft Windows betrachten. Eine gedruckte Version dieser Dokumentation findet sich in Kapitel 3 - außerdem ist der Quellcode selbst mit Anmerkungen und Kommentaren versehen.

Die Installation des objektorientierten Moduls wird nach der für Perl typischen Weise vorgenommen:

1. Entpacken des Archivs *Lingua-Tokenize-0.7.tar.gz*
2. Öffnen eines Terminals/der Eingabeaufforderung und wechseln in das durch das Entpacken des Archivs neu entstandene Verzeichnis
3. Erstellen des Makefiles durch Aufruf von `perl Makefile.PL`
4. „Bauen“ des Programms durch Aufruf von `make` [1]
5. Testen der korrekten Funktionsweise der Software durch Aufruf von `make test`
6. Nach erfolgreich abgeschlossenen Tests kann die Installation des Moduls mit dem Befehl `make install` abgeschlossen werden

Liste der Abhängigkeiten:

- Perl (Version 5.8.1 oder höher)
- Perl-Modul *Carp* (Standarddistribution)

Anmerkungen:
Dem Archiv liegen zusätzliche Skripte bei, mit deren Hilfe unter anderem eine exemplarische Satzextraktion aus einem gegeben Korpus auf der Kommandozeile durchgeführt

[1] Auf Microsoft Windows-Systemen wird dem Benutzer nach Wissen des Autors standardmäßig kein *make*-Tool zur Verfügung gestellt. Als Ersatz kann das (mit *Lingua::Tokenize* getestete) Tool *nmake* dienen, das u.a. auf dem FTP-Server des sprachwissenschaftlichen Instituts zur Verfügung steht (*programming/languages/perl/windows/*). In diesem Fall ist bei den Kommandos `make` durch `nmake` zu substituieren.

werden kann (*extract_sentences.pl*). Anhand des entstehenden Debugging-Outputs kann der Ablauf von Tokenisierung, Normalisierung und der anschließenden Segmentierung zu Satzeinheiten nachvollzogen werden. Zur exemplarischen Tokenisierung dient das Skript *tokenize.pl*.

```
C:\Tagger\Lingua-Tokenize-0.7>perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Lingua::Tokenize

C:\Tagger\Lingua-Tokenize-0.7>nmake
cp lib/Lingua/Tokenize.pm blib\lib\Lingua\Tokenize.pm
cp tokenize.pl blib\lib\Lingua\tokenize.pl
cp extract_sentences.pl blib\lib\Lingua\extract_sentences.pl

C:\Tagger\Lingua-Tokenize-0.7>nmake test
C:\Perl\bin\perl.exe "-MExtUtils::Command::MM" "-e" "test_harness(0,
    'blib\lib', 'blib\arch')" t\Lingua-Tokenize.t
t\Lingua-Tokenize....ok
All tests successful.
Files=1, Tests=4, 0 wallclock secs ( 0.00 cusr +  0.00 csys =  0.00 CPU)

C:\Tagger\Lingua-Tokenize-0.7>nmake install
Installing C:\Perl\site\lib\Lingua\Tokenize.pm
Appending installation info to C:\Perl\lib/perllocal.pod
```

Abbildung 1: Ausschnitt einer *Lingua::Tokenize*-Installation auf einem Windows-System

1.1.2 XEROX Finite State Tools

Die von **XEROX** entwickelten Finite State Tools müssen auf jedem System, auf dem der vorliegende Tagger zur Anwendung kommen soll, installiert sein. Betriebssystem-spezifische Installationsanweisungen werden mit der Software ausgeliefert.

Auf einem Windows-System ist es erforderlich, den Pfad zu kennen, unter dem die Binärdateien installiert wurden (z.B. *C:\Tagger\xerox-windows*). Dieser wird sowohl bei der nachfolgenden Installation des Perl-Moduls *XFST::Lookup* als auch bei der des Hauptprogramms *disambiguate.pl* benötigt.

1.1.3 XFST::Lookup

Das objektorientierte Perl-Modul *XFST::Lookup* stellt ein Interface zwischen Perl und dem von **XEROX** vertriebenen Finite State Tool *lookup* bereit, das es dem Benutzer möglich macht, unabhängig vom verwendeten Betriebssystem eine morphologische Analyse unter Rückgriff auf besagtes *lookup*-Tool durchführen zu können.

Die erzielte Abstraktion erlaubt es, der Objektmethode *lookup()* die notwendigen Parameter zu übergeben und einen *Hash of Arrays* zurückgegeben zu bekommen, der direkt weiterverarbeitet werden kann. In dieser komplexen Datenstruktur finden sich jeweils die konkreten Wortformen als *Key* mit einem Array der Kohorten aller möglichen Lesarten als *Value*.

Das Modul arbeitet in beiden von *lookup* zur Verfügung gestellten Modi, also sowohl mit einem einzelnen Transducer, als auch mit Lookup-Strategien und bietet die Möglichkeit, direkt aus Perl heraus Flags über die Objektmethode *set_flags()* zu setzen und somit die Funktionsweise der Analyse gezielt zu beeinflussen. Falls ein Anpassen der Flags an die Funktionsweise des Moduls notwendig sein sollte, führt *XFST::Lookup* dies eigenständig durch.

Eine ausführliche Dokumentation des Moduls liegt im Perl-eigenen *POD*-Format vor und lässt sich durch Absetzen des Kommandos `perldoc XFST::Lookup` auf der Kommandozeile einsehen. Eine gedruckte Version dieser Dokumentation findet sich in Kapitel 4 - des Weiteren finden sich weiterführende Kommentare im Quellcode des Moduls selbst.

Die folgenden Schritte sind notwendig, um das Modul zu installieren:

1. Entpacken des Archivs *XFST-Lookup-0.11.tar.gz*
2. Öffnen eines Terminals/der Eingabeaufforderung und wechseln in das durch das Entpacken des Archivs neu entstandene Verzeichnis
3. Erstellen des Makefiles durch Aufruf von `perl Makefile.PL`
4. „Bauen“ des Programms durch Aufruf von `make`
5. Auf Windows-Systemen kann es notwendig sein, den Pfad zu den von **XEROX** bereitsgestellten Finite State Tools (*lookup* im Speziellen) auf die folgende Weise zu spezifizieren: `set XFST_PATH=Pfad\zu\lookup` [2]
6. Testen der korrekten Funktionsweise der Software durch Aufruf von `make test`
7. Nach erfolgreich abgeschlossenen Tests kann die Installation des Moduls mit dem Befehl `make install` abgeschlossen werden

[2] Ein diesbezügliches Beispiel findet sich sowohl in der Fehlerdiagnose von `make test` als auch im Ausschnitt der Beispielinstallation (siehe Abbildung 1.1.3)

Liste der Abhängigkeiten:

- Perl (Version 5.8.1 oder höher)
- Perl-Modul *Carp* (Standarddistribution)
- lookup (**XEROX** Finite State Tools)

Anmerkungen:
Mit Hilfe des im Archiv enthaltenen Skripts *lookup.pl* kann eine morphologische Analyse andhand eines Inputs und eines einzelnen Transducers durchgeführt werden um sich mit der zurückgegebenen Datenstruktur vertraut zu machen, die im *Data::Dumper*-Format ausgegeben wird und um den Programmablauf anhand des Debugging-Outputs nachzuvollziehen.

```
C:\Tagger\XFST-Lookup-0.11>perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for XFST::Lookup

C:\Tagger\XFST-Lookup-0.11>nmake
cp lookup.pl blib\lib\XFST\lookup.pl
cp lib/XFST/Lookup.pm blib\lib\XFST\Lookup.pm

C:\Tagger\XFST-Lookup-0.11>set XFST_PATH=C:\Tagger\xerox-windows

C:\Tagger\XFST-Lookup-0.11>nmake test
C:\Perl\bin\perl.exe "-MExtUtils::Command::MM" "-e" "test_harness(0,
    'blib\lib', 'blib\arch')" t\XFST-Lookup.t
t\XFST-Lookup....ok
All tests successful.
Files=1, Tests=3, 0 wallclock secs ( 0.00 cusr +  0.00 csys =  0.00 CPU)
Stop.

C:\Tagger\XFST-Lookup-0.11>nmake install
Installing C:\Perl\site\lib\XFST\lookup.pl
Installing C:\Perl\site\lib\XFST\Lookup.pm
Appending installation info to C:\Perl\lib/perllocal.pod
```

Abbildung 2: Ausschnitt einer *XFST::Lookup*-Installation auf einem Windows-System

1.2 Installation der Transducer

Die der morphologischen Analyse zugrundeliegenden Transducer werden sowohl in binärer als auch in Form des Quellcodes bereitgestellt.

1.2.1 Benutzen der binären Transducer

Die binären Transducer sind betriebssystem- und architekturunabhängig, so dass sie gleichsam auf allen unterstützten Systemen zum Einsatz kommen können (siehe auch Punkt 6). Das Archiv *german-lexika-0.46-bin.tar.gz* enthält vier Transducer mit einer entpackten Gesamtgröße von etwa 4 MB und das diese verbindende Lookup-Strategie-Skript *strategy.txt*.

Das Archiv lässt sich mit allen herkömmlichen Packprogrammen dekomprimieren und entpacken, so z.B. mit den Tools *tar* und gzip (GNU/Linux), *StuffIt* (Mac OS X) und *WinZip* (Microsoft Windows).

Da zum Kompilieren des Quellcodes ein unixähnliches Betriebssystem notwendig ist, bietet es sich für Winodws-Benutzer an, auf die in binärer Form vorliegenden Transducer zurückzugreifen [3].

1.2.2 Kompilieren des Quellcodes

Nach Entpacken des Archivs lassen sich die etwa 18.000 Zeilen des Quellcodes mit Hilfe des beiliegenden *Makefiles* schnell und einfach in eine binäre Form bringen.

Benutzer, die den Pfad zu dem **XEROX** Finite State Tool *xfst* nicht in ihrer *$PATH*-Variable haben, müssen diese Umgebungsvariable dahingehend anpassen, z.B. wie folgt (Bourne-Again-Shell-Syntax):

```
export PATH="$PATH:/opt/xerox-linux/"
```

Alternativ ist auch im Makefile selbst eine diesbezügliche Möglichkeit vorgesehen. Dazu muss das Makefile editiert und die Variable XFST_PATH [4] der jeweiligen Konfiguration entsprechend angepasst werden, hier am Beispiel eines Mac OS X-Systems:

```
XFST_PATH=/opt/xerox-macosx
```

Anschließend sorgt ein Aufruf von `make clean` gefolgt von `make all` dafür, dass die 57 einzelnen Lexika kompiliert und anschließend in der richtigen Konstellation vereinigt werden. Die resultierenden Lexika sind wie folgt aufgeteilt:

Transducer	Funktion
german.fst	Hartkodierte Stamm-Affix-Kombinationen
guesser.fst	morphologisch mögliche Stamm-Affix-Kombinationen
last_resort.fst	Fallback: Zuweisen aller Tags
alternation.fst	Reguläre Ausdrücke zur Modellierung optionaler orthographischer Variationen

[3] Für Windowsbenutzer mag auch in dieser Hinsicht die Verwendung von *Cygwin* (`http://www.cygwin.com/`) interessant sein, denn unter *Cygwin* lassen sich mit *GNU make* sowohl die Lexika kompilieren als auch ist der Tagger im allgemeinen dort lauffähig.

[4] Die Deklaration der Variable findet sich gleich am Anfang des Makefiles, Zeile 3

```
$ make clean all
>>> Making clean in '/home/alex/Coding/cvsroot/tagger/lexicon'
>>> Building:
        [dirs]          creating base dirs
        [lexc]          ART.lexc -> ART.fst
        [lexc]          KON.lexc -> KON.fst
        [lexc]          KOUI.lexc -> KOUI.fst
        [lexc]          KOUS.lexc -> KOUS.fst
        [lexc]          KOKOM.lexc -> KOKOM.fst
        [lexc]          PUNCT.lexc -> PUNCT.fst
        [lexc]          APPO.lexc -> APPO.fst
        [lexc]          APZR.lexc -> APZR.fst
        [lexc]          PTKZU.lexc -> PTKZU.fst
        [lexc]          VMPP.lexc -> VMPP.fst
        [lexc]          PPER.lexc -> PPER.fst
        [lexc]          PRELAT.lexc -> PRELAT.fst
        [lexc]          PRELS.lexc -> PRELS.fst
        [lexc]          PWAT.lexc -> PWAT.fst
        [lexc]          PWS.lexc -> PWS.fst
        [lexc]          PRF.lexc -> PRF.fst
        [lexc]          VVIZU.lexc -> VVIZU.fst
        [lexc]          VVPP.lexc -> VVPP.fst
        [...]
        [script]        CARD.script -> CARD.fst
        [script]        NN.script -> NN.fst
        [script]        VVFIN.script -> VVFIN.fst
        [script]        VVIMP.script -> VVIMP.fst
        [script]        ADJA.script -> ADJA.fst
        [script]        PAV.script -> PAV.fst
        [script]        VVINF.script -> VVINF.fst
>>> Building target 'german.fst'
        [guesser]       TRUNC.script -> TRUNC.fst
        [guesser]       NNg.script -> NNg.fst
        [guesser]       VVFINg.script -> VVFINg.fst
        [guesser]       ADJDg.script -> ADJDg.fst
        [guesser]       ADJAg.script -> ADJAg.fst
>>> Building POS-guesser 'guesser.fst'
        [fallback]      FALLBACK.script -> FALLBACK.fst
>>> Building last resort fallback-guesser 'last_resort.fst'
        [alternation]   ORTHOGRAPHY.script -> ORTHOGRAPHY.fst
>>> Building alternations 'alternation.fst'
>>> All done.
```

Abbildung 3: Ausschnitt der Kompilierung der Transducer

Auf langsameren Systemen bietet es sich an, bei Veränderungen am Quellcode nicht immer die gesamten Transducer zu kompilieren (besonders das Generieren des Guesser-Netzwerks nimmt einige Zeit in Anspruch!), sondern nur das erforderliche Netzwerk neu zu kompilieren. Dazu kann der Name des jeweiligen Transducers ohne das Suffix *„.fst“* als Parameter an `make` übergeben werden, z.B. also:

```
make guesser
```

Um eine Distribution der Lexika zu Erzeugen, z.B. um eigene Veränderungen Anderen zur Verfügung stellen zu können, kann auf das Kommando `make dist` zurückgegriffen werden - es empfiehlt sich unter diesen Umständen auch die Makefilevariable *VERSION* zu modifizieren.

1.2.3 Installation der Lexika

Es wird davon ausgegangen, dass der Benutzer ein beliebiges Verzeichnis für die Installation der Software gewählt hat - auf dieses wird im Folgenden mit *INSTDIR* referiert. Die folgenden Dateien müssen nach *INSTDIR* kopiert werden, um die Software auf eine Verwendung vorzubereiten:

- *german.fst*
- *guesser.fst*
- *last_resort.fst*
- *alternation.fst*
- *strategy.txt*

1.3 Installation des Hauptprogramms

Das Hauptprogramm *disambiguate.pl* führt unter Rückgriff auf die Funktionalität der Module *Lingua::Tokenize* und *XFST::Lookup* und der morphologischen Analyse die aus den Lexika generierten Transducer die eigentliche Disambiguierung durch. Da es sich um ein Perl-Programm handelt, genügt es, dieses nach *INSTDIR* zu kopieren. Allerdings muss darüber hinaus ggf. der Pfad zu den **XEROX**-Binaries angepasst werden, sollten diese sich nicht im Standardsuchpfad des jeweiligen Betriebssystems befinden.

Ausführliche Informationen zur Funktionsweise von *disambiguate.pl* finden sich unter Punkt 2.

1.4 Installation der Constraints

Als letztes müssen die Constraints, also die syntaktischen Disambiguierungsregeln, noch integriert werden. Dazu genügt es, das Archiv *Constraints.tar.gz* in *INSTDIR* zu entpacken, damit *disambiguate.pl* Zugriff auf die einzelnen Textdateien hat.

1.5 Anmerkung

Das ebenfalls zur Verfügung stehende Archiv *tagger.tar.gz* enthält alle nicht modularisiserten Programmteile des Taggers, es genügt also auch, die Perl-Module *Lingua::Tokenize* und *XFST::Lookup* samt deren Abhängigkeiten (**XEROX** Finite State Tools) zu installieren und ansonsten auf dieses Archiv zurückzugreifen. Es enthält sowohl die Constraints, als auch die (binären) Transducer samt Lookup-Strategie-Skript und das Hauptprogramm *disambiguate.pl*.

2 Disambiguierung: disambiguate.pl

Das vorliegende Programm und die dazu gehörige Umgebung ist ein Prototyp, welcher die Realisierung einer Constraint-Grammatik am Beispiel des Deutschen, bzw. eines Auszuges des Deutschen, demonstrieren soll. Es ist jedoch so konzipiert, dass es prinzipiell für jede Sprache, auf die Constraint-Grammatiken anwendbar sind, benutzbar ist, da die benötigten Constraints und Lexika vom Programmcode getrennt sind.

Anhand der Implementierung soll gezeigt werden, wie mit grammatischen Regeln vorher erzeugte morphologische Analysen zu in einem Korpus vorkommenden Wortformen disambiguiert werden können, also ungrammatische Lesarten aus den durch eine solche morphologische Analyse erzeugten Kohorten entfernt werden können.

Die Datei *disambiguate.pl* ist ein Perl-Programm, welches zum einen den Rahmen und die Struktur des Disambiguierungsprozesses inklusive der vorhergehenden Tokenisierung und morphologischen Analyse abbildet und zum anderen an eigentlicher Logik den Disambiguierungsvorgang der einzelnen Kohorten implementiert.

Das Programm benötigt die Pakete *XFST::Lookup* und *Lingua::Tokenize* und stellt selbst die im Folgenden beschriebenen Funktionen zur Verfügung.

2.1 Funktionen

2.1.1 disambiguate()

Signatur
Diese Funktion nimmt folgende Argumente:

1. @thisSentencesToBeDisambiguated: die zu disambiguierenden Sätze als Array
2. %thisCohorts: die Kohorten der Wortformen als Hash
3. $hashMode: ein Flag, das angibt, ob die Kohorten nach Worten(=0) oder nach ihrer Position im Korpus indiziert werden

Sie gibt eine Hash-Referenz zurück, die die disambiguierten Kohorten, indiziert nach ihrer Position im Korpus, enthält (%disambiguatedCohorts).

Funktionalität
Diese Funktion übernimmt die eigentliche Logik des Disambiguierens, indem sie anhand der im Verzeichnis *./constraints/* als Textdateien abgelegten Constraints die Lesarten verwirft, welche laut der Constraint-Grammatik, die durch diese Constraints konstituiert wird, an der gegebenen Position auf Grund von bestimmten Umgebungsbedingungen ungrammatisch sind.

Dabei werden die Constraints entsprechend verschiedener Modi verarbeitet, diese sind im einzelnen [5]:

• DISCARD_THIS
Dieser Modus verwirft die behandelte Lesart, wenn die im Constraint angegebenen Bedingungen erfüllt sind.
• DISCARD_OTHERS
Dieser Modus verwirft alle anderen als die behandelte Lesart, wenn die im Constraint angegebenen Bedingungen erfüllt sind.
• DISCARD_STRICTLY
Dieser Modus verwirft die behandelte Lesart, wenn die im Constraint angegebenen Bedingungen erfüllt sind und verwirft alle anderen als die behandelte Lesart, wenn diese nicht erfüllt sind.
• CHANGE_THIS
Dieser Modus verändert die behandelte Lesart und entspricht einer im Constraint anzugebenden Zeichenkette, wenn die im Constraint angegebenen Bedingungen erfüllt sind.

Des Weiteren gibt es noch zu jedem Modus eine Variante mit dem Suffix „_UNLESS“. Diese Modi entsprechen der jeweiligen Komplementrelation, also wird z.B. bei DISCARD_THIS_UNLESS die behandelte Lesart genau dann verworfen, wenn die Bedingung *nicht* erfüllt ist:

• DISCARD_THIS_UNLESS
• DISCARD_OTHERS_UNLESS
• DISCARD_STRICTLY
• CHANGE_THIS_UNLESS

2.1.2 loadDisambiguatedCohorts()

Signatur
Diese Funktion gibt eine Hash-Referenz mit den disambiguierten Kohorten zurück (%thisDisambiguatedCohorts).

Funktionalität
Diese Funktion lädt die mittels disambiguate() disambiguierten Kohorten aus der Datei

[5] die Dokumentation des exakten Constraint-Formats erfolgt unter 2.3

disambiguated-cohorts.txt im Arbeitsverzeichnis des Programms. Sie wird nicht mehr benötigt, da die Kohorten mittlerweile als Return-Werte zurückgegeben werden.

2.1.3 saveDisambiguatedCohorts()

Signatur
Diese Funktion nimmt eine Hash-Referenz mit den disambiguierten Kohorten als Argument (%thisDisambiguatedCohorts).

Funktionalität
Diese Funktion speichert die mittels disambiguate() disambiguierten Kohorten in der Datei *disambiguated-cohorts.txt* im Arbeitsverzeichnis des Programms.

2.1.4 readConstraints()

Signatur
Diese Funktion gibt eine Hash-Referenz mit allen eingelesenen Constraints zurück (%constraints).

Funktionalität
Diese Funktion sorgt dafür, dass alle Constraints aus dem Verzeichnis *./constraints/* aus den entsprechenden Textdateien eingelesen und in einem Hash gespeichert und zurückgegeben werden.

2.1.5 readSentences()

Signatur
Diese Funktion nimmt die im Korpus enthaltenen Token (@thisToken) als Argument und gibt eine Array-Referenz mit Sätzen (@sentences) zurück.

Funktionalität
Diese Funktion wurde im Anfangsstadium des Programms dazu benötigt, aus den einzelnen Token eines Korpus die Sätze zusammenzusetzen, mittlerweile wird dies aufgrund struktureller Überlegungen im Tokenizer gekapselt und mittels

```
$tokenizer->extract_sentences()
```

ausgeführt.

2.1.6 displayCohorts()

Signatur
Diese Funktion nimmt folgende Argumente:

1. %thisCohorts: die Kohorten der Wortformen als Hash
2. @thisSentences: die Sätze als Array

3. $displayMode: ein Flag, das angibt, ob die Kohorten nach Worten (=0) oder nach ihrer Position im Korpus indiziert werden

Funktionalität
Diese Funktion stellt die Sätze eines Korpus und die zu den darin enthaltenen Token gehörigen Kohorten, entweder in disambiguierter, oder in nicht disambiguierter Form dar. Sie wird zugunsten der neuen Funktion displayCohortsInTable() nicht mehr benutzt, da die Darstellung von zu disambiguierenden und disambiguierten Kohorten zu unübersichtlich war.

2.1.7 displayCohortsInTable()

Signatur
Diese Funktion nimmt folgende Argumente:

1. %thisCohorts: die zu disambiguierenden Kohorten der Wortformen als Hash
2. %thisDisambiguatedCohorts: die disambiguierten Kohorten der Wortformen als Hash
3. @thisSentences: die Sätze als Array

Funktionalität
Diese Funktion stellt die Sätze eines Korpus und die zu den darin enthaltenen Token gehörigen disambiguierten und nicht disambiguierten Kohorten in tabellarischer Form dar.

2.2 Parameter

$disambiguationIterations (Zeile 52):
Anzahl der Iterationsdruchläufe, Standardwert: 4

$corpusFile (Zeile 55):
Pfad und Dateiname der Korpusdatei, Standardwert: *testsatz.txt*

$xfstPath (Zeile 58):
Pfad zum XEROX XFST, Standardwert: */opt/xerox-macosx/*

2.3 Format der Constraints

Die Constraints müssen als Textdatei im Verzeichnis *./constraints/* abgelegt werden, die Datei muss dabei dieser Namenskonvention entsprechen:

LESART_AUF_DIE_DIESE_CONSTRAINTS_ANGEWENDET_WERDEN_SOLLEN.txt

In den einzelnen Textdateien stehen nun einer oder mehrere Constraints, jeweils durch einen Zeilenumbruch getrennt. Die Constraints haben dabei folgendes Format:
MODUS[=x] y/CONDITION

MODUS nimmt dabei den Wert der in 2.1 beschriebenen Modi an, falls der Modus CHANGE_THIS oder CHANGE_THIS_UNLESS lautet, muss der Modusangabe noch ein „=“ angefügt werden, gefolgt von der Zeichenkette, in die die betreffende Lesart geändert werden soll. y gibt die Position an, auf die sich diese Bedingung bezieht, ein negativer Wert bedeutet, jeweils auf den aktuellen Satz bezogen, etwas links von der aktuellen Position und ein positiver Wert rechts von der aktuellen Position. CONDITION enthält nun die Lesart, die an der angegebenen Position vorkommen soll, damit die Bedingung erfüllt ist, hierbei sind Wildcards (* bzw. Kleene-Star) erlaubt.

```
DISCARD_OTHERS 1/VVFIN+PRAES+3P+SG
DISCARD_OTHERS 1/VAFIN+3P+SG
DISCARD_OTHERS 1/VMFIN+3P+SG
DISCARD_THIS 1/VVFIN+PRAES+2P+*
DISCARD_THIS 1/VAFIN+2P+*
DISCARD_THIS 1/VMFIN+2P+*
DISCARD_THIS 1/VVFIN+PRAES+1P+*
DISCARD_THIS 1/VAFIN+1P+*
DISCARD_THIS 1/VMFIN+1P+*
DISCARD_THIS 1/VVFIN+PRAES+3P+PL
DISCARD_THIS 1/VAFIN+3P+PL
DISCARD_THIS 1/VMFIN+3P+PL
DISCARD_THIS 1/PPER+*+*
DISCARD_THIS 1/N+*+*
```

Abbildung 4: Auszüge aus der Datei *./constraints/PPER+3p+SG+NOM.txt*

2.4 Weitere Informationen

Weitere Informationen, z.B. detailierte Angaben zu einzelnen Programmzeilen, sind im Programmcode selbst kommentiert.

3 POD-Dokumentation: Lingua::Tokenize

3.1 NAME

Lingua::Tokenize - Perl extension for Tokenization (and Normalization)

3.2 SYNOPSIS

```
use Lingua::Tokenize;

my $tokenizer = new Lingua::Tokenize( { debug => 1 } );

my @tokens = $tokenizer->tokenize( {
        normalize => 1,
        tokendef => 'words',
        input => \$wholeCorpusInAScalar
} );

my @types = $tokenizer->unify( \@tokens );

my @sentences = $tokenizer->extract_sentences( \@tokens );
```

3.3 DESCRIPTION

Lingua::Tokenize provides all methods necessary to tokenize and normalize **german** text from within Perl.

3.4 METHODS

3.4.1 unify() [\@arrayref]

unify() returns all types contained in an \@arrayref of tokens - either as an hash (providing occurence counts as values) or as an array.

As usual, the return-value is determined by the context unify() is called from.

**NOTE: at the moment, only an @array is returned :(

3.4.2 tokenize() { anonymous => hash }

tokenize() segments a given string of input into single elements called tokens and returns all of them in array-context.

the following options may be supplied:

normalize [bool]

Shall the input be normalised before tokenization?

Please NOTE: Only german normalization-rules are implemented at the moment.

input [\$scalar-ref]

Set the input that shall be tokenized. Must be given as scalar-reference.

tokendef ['words'|'sentences']

setting tokendef to *'sentences'* will behave like a shortcut: all tokens will be passed to extract_sentences() and the sentences returned instead of the word-tokens.

3.4.3 extract_sentences()

extract_sentences() extracts all sentences from a given arrayref and returns them as an array of sentences. in order to gain appropriate results it is important to pass an array that was generated by tokenize().

Here's an example:

```
my @sentences = $tokenize->extract_sentences( \@wordtokens );
```

3.5 SEE ALSO

[1] "What is a word, What is a sentence? Problems of Tokenization",
Gregory Grefenstette, Pasi Tapanainen

Visit "http://homepage.rub.de/Alexander.Linke-2/"for updates.

3.6 AUTHORS

Alex Linke, <Alexander.Linke-2@rub.de>
Rona Linke, <Rona.Linke@rub.de>

3.7 COPYRIGHT AND LICENSE

4 POD-Dokumentation: XFST::Lookup

4.1 NAME

XFST::Lookup - Perl extension to access *XEROX*'s lookup-utility

4.2 SYNOPSIS

```
use XFST::Lookup;
my $lookup = new XFST::Lookup({debug=>1);

$lookup->set_flags("some", "flags");

# to use a single FST:
my %looked_up_wordforms = $lookup->lookup({
    fst => 'path/to_a/finite_state_transducer',
    words => \@wordforms_as_array_REFERENCE
} );

# to use a lookup-strategy:
my %looked_up_wordforms = $lookup->lookup({
    strategy => 'path/to_a/lookup_strategy',
    words => \@wordforms_as_array_REFERENCE
} );
```

4.3 DESCRIPTION

XFST::Lookup() provides an easy way to access the functionality of the *XEROX* lookup-utility from within Perl. *lookup* itself "applies a pre-compiled transducer or transducers to look up words".

4.4 CONSTRUCTOR

You may provide parameters via an anonymous hashref like this:

```
my $lookup = new XFST::Lookup( {
        debug => 1 ,
        flags => 'some flags',
        path => '/home/alex/bin/:/home/roni/bin/',
        fst => 'path/to_a/FST/file'
} );
```

debug [BOOL]

turns very verbose debugging messages on

flags [$scalar]

set lookup-flags (see set_flags())

fst [path]

set the path to a finite state transducer _once_ this way, you won't have to set it every time you use lookup()

strategy [path]

set the path to the file containing your lookup-strategy _once_ this way, you won't have to set it every time you use lookup()

path [path]

*Linux/*BSD/MacOS*:

Specify a single addition or a list of additions to $PATH in order to help XFST::Lookup to locate the lookup-binary. (separator: ':')

Per default XFST::Lookup is looking for the binary in $ENV{PATH} on Linux,*BSD and Mac OS calling the which(1)-utility that is shipped with every *NIX-distribution.

Win32:

On MSWin32 XFST::Lookup tries to locate the binary in the defaut path or searches a user-defined path afterwards (i.e. supplied via "path=>d:\alex\xerox-windows\")

Manually setting the path on this OS is recommended...

4.5 METHODS

4.5.1 set_flags() [@array|$scalar]

set_flags() is used to set the flags that will be used by *XEROX*'s lookup-utility. See [3] for a brief overview.

Please note: even if you set them manually, the 'LxL' and 'TxT'-flags are stripped, in order to do what you mean. XFST::Lookup needs the default separators in order to correctly process the output of lookup.

4.5.2 lookup() [{anonymous hashref}]

... is the most important method of XFST::Lookup (like the name may already suggest). Every time lookup() is invoked from the inside of a Perl-program, *XEROX*'s lookup-utility is called with the given options and all results are returned as a *hash of arrays*. This datastructure has been chosen because there may be ambiguous word-forms so a simple hash won't fulfill the needs.

Here's an example of the mentioned datastructure (Data::Dumper-format):

```
    $VAR1 = 'eine';
    $VAR2 = [
              'ein+ART+NOM+SG',
              'ein+ART+AKK+SG',
              'ein+PIS+AKK+SG',
              'ein+PIS+NOM+SG',
              'ein+PIAT+AKK+SG',
              'ein+PIAT+NOM+SG',
              'eine+PIDAT'
            ];
    $VAR3 = 'Haus';
    $VAR4 = [
              'Haus+NN+NOM+SG',
              'Haus+NN+DAT+SG',
              'Haus+NN+AKK+SG'
            ];
    $VAR5 = 'das';
    $VAR6 = [
              'das+ART+NOM+SG',
              'das+ART+AKK+SG',
              'das+KOUS',
              'der+PDAT+NOM+SG',
              'der+PDAT+AKK+SG',
              'der+PDS+NOM+SG',
              'der+PDS+AKK+SG'
            ];
    ...
```

While the hash's key contains a *wordform*, the value provides an @array of all analyzed *readings* of this wordform.
The following parameters are available to alter XFST::Lookup's runtime behaviour:

fst [path]

set the path to the file that contains the finite state transducer to use ('save stack'-format).

fst is optional at this point iff it was already passed to the constructor, it is forbidden if you use the *strategy*-parameter at the same time.

Please note: Setting **fst** again at this point will overwrite the setting for the current run _only_. see CONSTRUCTOR

strategy [path]

set parameter to the path to the strategy-file if you want to use a lookup-strategy rather than a single finite state transducer.

Conflicts with the *fst*-parameter.

Please note: Setting **strategy** again at this point will overwrite the setting for the current run _only_. see CONSTRUCTOR

words [\@array_REFERENCE]

this parameter is _obligatory_ and has to contain an *array-reference* to a list of wordforms that will be analyzed by *XEROX*'s lookup-utility.

4.6 SEE ALSO

[1] "*Finite State Morphology*", Kenneth R. Beesley and Lauri Karttunen, CSLI Publications Stanford
[2] http://www.fsmbook.com
[3] the output of 'lookup -h'
[4] updates of XFST::Lookup may be available at http://homepage.rub.de/Alexander.Linke-2

4.7 AUTHOR

Alex Linke, <Alexander.Linke-2@ruhr-uni-bochum.de>

4.8 COPYRIGHT AND LICENSE

5 Lizenzen

Die Perl-Module *XFST::Lookup* und *Lingua::Tokenize* werden der Öffentlichkeit unter den gleichen Lizenzbedingungen wie Perl selbst bereitgestellt - alle Lexika sowie das Hauptprogramm *disambiguate.pl* stehen unter den Bedingungen der *GNU General Public License* (Version 2) [6] zur Verfügung.

6 Getestete Betriebssysteme

Das Programm wurde unter den folgenden Betriebssystemen auf korrekte Funktionsweise getestet:

- GNU/Linux (*Gentoo*)
- Mac OS X (*„Panther“* (10.3), *„Tiger“* (10.4))
- Microsoft Windows 2000 NT
- Microsoft Windows XP
- FreeBSD (5.4-RELEASE) (unter Linux-Emulation) [7]
- Cygwin (1.5.18-1)

7 Aufteilung der Gesamtaufgaben

Der Constraint-Tagger ist in *Teamarbeit* entstanden! Die folgende Tabelle enthält eine Übersicht darüber, wer sich auf welche Teilaufgaben konzentriert hat:

Stefanie Konetzka	Constraints
Alexander Linke	Lexika, *Lingua::Tokenize*, *XFST::Lookup*, Installationsanleitung, Moduldokumentation
Rona Linke	Lexika, *Lingua::Tokenize*, Dokumentation
Björn Wilmsmann	*disambiguate.pl* (Hauptprogramm), Dokumentation „Disambiguerung“, Entwurf und Dokumentation des Constraint-Formats

[6] Siehe u.a. `http://www.gnu.org/licenses/gpl.html`

[7] Zum Kompilieren der Lexika muss *GNU make* (gmake) anstelle von *pmake* verwendet werden.