

Digitalisierung und Aufbereitung vorhandener Forschungsdaten

Am Beispiel des Angguruk-Wörterbuchs



Ludwig-Maximilians-Universität München

Andreas Neumann

Institut für Ethnologie und Afrikanistik

Münchner Str. 7

Hauptseminar: Dokumentation. Sprachen,
Texte, Gespräche

85635 Höhenkirchen

Tel. 0176 / 220 50 984

Abstract

Ziel der Arbeit ist es aus vorhandenen Forschungsdaten eine digitale Repräsentation zu gewinnen. Das auf gewonnene Digitalisat soll den folgenden Ansprüchen genügen:

Es soll archivierbar und verteilbar sein.

Es soll einfach und verständlich sein.

Es soll maschinenlesbar sein.

In der Arbeit wird ein Zielformat entwickelt, auf Probleme bei der Übertragung der alten Daten in dieses Format eingegangen. Im letzten Kapitel werden Anwendungsmöglichkeiten die sich aus der Digitalisierung der Daten ergeben gezeigt.

Inhalt

1. Digitalisierung und Archivierung von Daten	4
1.1 Metainformationen.....	4
1.2 Strukturierung der Daten.....	4
1.3 Verteilbarkeit.....	4
2. Die Ausgangsdaten	5
2.1 Das verwendete Dateiformat.....	5
2.2 Logischer Aufbau der Ausgangsdaten.....	6
2.2.1 Eintragstypen.....	6
2.2.2 Aufbau eines Eintrags.....	9
3. Das Zielformat	10
3.1 XML.....	10
3.2 Aufbau des Zielformats.....	10
3.3 Dokumententypdefinition (DTD).....	12
3.4 Darstellung und Umformung der Daten mit Cascading-Style-Sheets.....	13
3.5 Beispiel für einen Eintrag.....	14
4. Vorgehen	16
4.1 Extrahieren der semi-strukturierten Daten.....	16
4.2 Probleme.....	16
4.2.1 Uneinheitlicher Aufbau der Einträge.....	16
4.2.2 Steuerungszeichen wurden zur Formatierung genutzt.....	17
4.2.3 Uneinheitliche Übersetzung.....	17
5. Neue Möglichkeiten	18
5.1 Ein Wörterbuch.....	18
5.2 Ein Korpus.....	19
5.3 Ein primitives Übersetzungsprogramm.....	19
5.4 Aufspüren von Lücken im Wörterbuch mit Hilfe eines Korpus.....	20
6. Fazit	21
Anhang	23
1. x_ang.rb.....	23
2. trans.rb.....	26
3. look_for_missing_words.rb.....	27

I. Digitalisierung und Archivierung von Daten

Im Rahmen des Hauptseminars "Dokumentation. Sprachen, Texte, Gespräche" unter der Leitung von Volker Heeschen, zeigte sich, dass die Ansprüche die eine moderne Archivierung an ein Dokument stellt, sich stark von der klassischen Idee des Dokuments, das als Manuscript in einer Bibliothek liegt, abweicht.

Das Gewicht liegt nun vor allem auf Metainformationen, Strukturierung der Daten und leichter Verteilbarkeit.

Damit nun vorhandene Forschungsdaten nicht in alten Archiven verstauben, weil sie schlichtweg nicht wahrgenommen werden oder der Aufwand des Zugriffs zu hoch ist, ist es nötig sie in neue Formate zu überführen.

I.1 Metainformationen

Durch die ständig wachsende Datenmenge wird es immer wichtiger einen Text zusätzlich mit Metainformationen zu versehen. Sie erst ermöglichen das Auffinden von Dokumente in einer großen Menge von Texten.

I.2 Strukturierung der Daten

Nur aus strukturierten Daten ist es möglich maschinell Informationen zu extrahieren. Diese Strukturierung ist weniger inhaltlicher Natur, sondern eine Eigenschaft des gewählten Dateiformats.

I.3 Verteilbarkeit

In der Möglichkeit einfach und kostengünstig Kopien eines Dokuments zu erstellen und diese in sekundenschnelle weltweit zu verteilen liegt der Hauptvorteil der "neuen" (digitalen) Archi-

Wie unschwer zu erkennen ist, kann ein menschlicher Benutzer aus dieser Ausgabe kaum Informationen gewinnen oder auf den Inhalt der Datei schließen.

Ein weiteres, aus dem Dateiformat resultierendes, Problem ist die fehlende Robustheit. Sollte die Datei beschädigt werden oder der Übertragungsweg instabil sein, ist eine Rekonstruktion des Inhalts fast unmöglich.

Das schwerwiegendste Problem, das aus der Wahl des falschen Dateiformats resultiert, ist, dass keine Metainformation gespeichert werden und die innere Struktur der Daten nicht der Oberflächendarstellung entspricht.

Ein Eintrag ist somit nur, wenn überhaupt, aufgrund der grafischen Aufmachung als solcher zu identifizieren. Der Sinn der Untereinheiten muss aus dem Kontext erschlossen werden. Dieser Umstand verhindert das automatische Überführen in andere Dateiformate zur Sicherung, Verbreitung und Präsentation.

2.2 Logischer Aufbau der Ausgangsdaten

Interpretiert man als menschlicher Betrachter die Ausgangsdaten lassen sich vier Eintragstypen erkennen.

2.2.1 Eintragstypen

Typ I: Einfacher Eintrag

Dieser Typ besteht aus einem Stichwort gefolgt von einer Übersetzung.

Bsp.:

abit jabuk ein im Wald angelegter Garten ohne Beete
 Abit jabuk witukmv jet-ysarug lage.
 Wir sehen Leute, die einen Garten im Wald anlegen.

(Ziegler: S.1)

Typ 2: Eintrag mit mehreren Übersetzungen

Bei diesem Typ existieren für ein Stichwort mehrere Übersetzungsmöglichkeiten. Das Stichwort wird einmal genannt, darauf folgen nummeriert die möglichen Übersetzungen.

Bsp.:

- amlogo** (n)
amlogoki (pl.)
- 1) Söhne
 - 2) Söhne der Geschwister
 - 3) Söhne von Vaters Schwester (**vwan**)
 - 4) jüngere Brüder des Ehemannes

(Ziegler: S. 18)

Typ 3: Homonyme

Dieser Typ deckt Wörter ab, deren Bedeutung ein so großes Feld umfasst dass man von zwei getrennten Stichwörtern mit zufällig gleicher Schreibung ausgehen muss. Der Eintrag folgt folgendem Aufbau: Ziffer, Stichwort, Übersetzung.

Bsp.:

- 1) **andut**
ik andut ein Tau oder ein Holzstamm, den man quer über den Fluß legt,
um einen Kuskus darauf zu fangen
Aben pak wagarikim wabul vlvg hele ik fvman ik fvma nynydygyptuk. Nyndygyptvg pak ik andut famen watuk.
Wenn die Leute einen Kuskus ankommen sehen, werfen sie ein Tau von der einen Flußseite auf die andere, um ihn zu schießen. Wenn sie das getan haben, schießen sie ihn auf dem Tau.
- 2) **andut**
ik andut ein Becken, in dem sich Wasser sammelt, jetzt auch für unsere Waschbecken gebraucht.
Ik hvi andutmv wamburuk.
Am Fuße des Wasserfalles sammelt sich das Wasser in einem kleinen Becken.
- 3) **andut** das strohige Innere von labu, hvbvt, humi, kuluk, auch das Innere des Baumstammes des uluwa und pisi.
- 4) **andut**
ajeg andut Zahnfleisch

(Ziegler: S. 20)

Typ 4: Aufzählungen / Listen

Der vierte Typ wird verwendet um Wortlisten abzubilden.

Bsp.:

e	Baum, Holz		
<u>Verschiedene Baumarten:</u>			
e wali (für Bretter)		e jag	(Feuerholz)
e pange (Feuerholz)			e marapna
(Feuerholz)			
e pindip (Feuerholz)			e tyhyntyg
(Harz, für die Haare gebraucht			
e wyn (Feuerholz)			e somuk
(Feuerholz)			
e poret (Feuerholz)			e ka
(Feuerholz)			
e sylmy (Bretter)			e simbalo
(Feuerholz)			
e nongo (Feuerholz)			e sijehyry
(Feuerholz)			
e soluk (junge Blätter und Früchte eßbar)		e nvmba	(Bretter)
e wamfvmv (Feuerholz)			e hvbvt
(Feuerholz, Blüte wird von			
den Vögeln gefressen)			
e samle (Feuerholz)			e pabol
(Feuerholz)			
e svwalili (Feuerholz)			e sali
(Frauen schmücken sich mit e paby		(Feuerholz)	
dem schwarzen Fruchtsaft) ¹⁰			
e sylyp (Feuerholz)		e hog	(Feuerholz)
e wip (Holzpfeiler)		e song	(Feuerholz)
e songgalkal (Feuerholz)			e hamkal
(Feuerholz)			
e faluk (Feuerholz)			

(Ziegler: S.27)

2.2.2 Aufbau eines Eintrags

Kurz zusammengefasst:

- Ein Eintrag besteht stets aus einem Stichwort und einer Übersetzung.
- Optional folgt hinter dem Stichwort eine Kategorieangabe. (z.B.: n,pl).
- Darauf folgen optional Deklinationsformen.
- Optional folgen mit dem Stichwort gebildete Phrasen.
- Optional werden noch Beispielsätze mit Übersetzung angegeben.
- Über einen Eintrag verteilt können Verweise zu anderen Einträgen an beliebige Stelle auftreten.

3. Das Zielformat

Dies ist das Endformat in dem die Daten nach der Umformung vorliegen werden. Wichtig ist es hierbei die Intention des ursprünglichen Autors beizubehalten.

3.1 XML

Als Zielformat wird die "Extensible Markup Language", kurz XML verwendet. Unter anderem empfiehlt auch Peter K. Austin die Verwendung dieses Formats (Austin 2006: S. 101) XML bietet folgende Vorteile:

- Die Daten werden im Klartext gespeichert, d.h. ein menschlicher Leser kann die Daten ohne spezielle Software verstehen und bearbeiten.
- Das Format wird weltweit in den verschiedensten Bereichen eingesetzt und ist verbindlich standardisiert. (W3C 2006)
- Logische Zusammenhänge lassen sich explizit speichern und von grafischer Darstellung getrennt speichern. Dies führt zu ungeahnter Flexibilität im Bereich der Darstellung
- Durch das hinzufügen von Dokumententypdefinition ist es möglich eine eigene Grammatik in das Dokument zu integrieren die, die Konsistenz gewährleistet.

3.2 Aufbau des Zielformats

Das Zielformat soll im Aufbau der intendierten Logik des originalen Verfasser des Ausgangsmaterials wiedergeben. Folgende Elemente sind im Ausgangsformat enthalten:

<dic>

Wurzelement der Datei. Es kann mehrere Einträge des Typs <eintrag> enthalten.

<eintrag>

Unter diesem Tag wird jeweils ein Eintrag zu einer Einheit zusammengefasst. Er umfasst immer ein Stichwort <stichwort> und eine Übersetzung <trans>. Optional enthält er folgende Typen <trans2>, <satz>, <verweis>, <dekl> und <cat>.

<trans>

Hier findet sich die Deutsche Übersetzung für das Stichwort.

<cat>

Existiert für das Stichwort eine Anmerkung zur Kategorie findet sich dies unter <cat>.

<trans2>

Trans2 dient dazu um kurze Beispielphrasen mit Übersetzung aufzunehmen. Es besteht aus <t2_stichwort> und <t2_trans>. Wobei ersteres eine Phrase auf Angorruk und zweiteres die dazugehörige Übersetzung aufnimmt.

<satz>

Satz dient dazu ganze Beispielsätze aufzunehmen. Es enthält stets <satz_eintrag> und <satz_trans>, wobei <satz_eintrag> den Beispielsatz auf Angorruk enthält und in <satz_trans> die deutsche Übersetzung zu finden ist.

<dekl>

Dekl nimmt gefundene Deklinationsformen auf.

<verweis>

Unter <verweis> finden sich Querverbindungen zu anderen Einträgen.

3.3 Dokumententypdefinition (DTD)

Die Dokumententypdefinition ist Teil des XML-Dokuments und sorgt für eine einheitlich Form der Einträge. Das Dokument wird nur angezeigt, wenn die Anforderungen der DTD erfüllt sind. Alternativ ließe sich auch XSL oder eine andere Schemasprache verwenden.

```
<!ELEMENT dic (eintrag+)>
<!ELEMENT eintrag ((stichwort,verweis?,trans),(dekl*|trans2*|satz*|cat?)*)>
<!ELEMENT stichwort (#PCDATA)>
<!ELEMENT trans (#PCDATA)>
<!ELEMENT dekl (#PCDATA)>
<!ELEMENT cat (#PCDATA)>
<!ELEMENT verweis (#PCDATA)>
<!ELEMENT satz (satz_eintrag,satz_trans)>
<!ELEMENT satz_eintrag (#PCDATA)>
<!ELEMENT satz_trans (#PCDATA)>
<!ELEMENT trans2 (t2_stichwort,t2_translation)>
<!ELEMENT t2_stichwort (#PCDATA)>
<!ELEMENT t2_translation (#PCDATA)>
```

In normaler Sprache Ausgedrückt sagt die DTD folgendes:

- Jedes Element vom Typ <dic> enthält mindestes einen Eintrag vom Typ <Eintrag>.
- Jedes Element vom Typ <eintrag> enthält genau ein Element <stichwort> und ein Element <trans>. Dazwischen darf optional ein <verweis> auftauchen. Die Elemente <dekl>,<trans2>,<satz> und <cat> können, müssen aber nicht, beliebig häufig in beliebiger Reihenfolge auftreten oder auch gar nicht.
- Die Elemente <trans2> und <satz> müssen jeweils genau einmal ihre zwei Unterelemente <satz_eintrag>, <satz_trans> bzw. <t2_stichwort>,<t2_translation> enthalten.
- Jedes nicht genannte Element darf mit beliebigen Textinhalt gefüllt werden.

3.4 Darstellung und Umformung der Daten mit Cascading-Style-Sheets

Zur Darstellung der Daten werden Cascading-Style-Sheets (CSS) benutzt. Mit Hilfe von CSS ist es möglich festzulegen wie logische Tags z.B. in einem Webbrowser dargestellt werden. Durch die Benutzung verschiedener Stylesheets ist es möglich aus einer einzigen XML Datei verschiedene Ausgabeformen zu generieren. Höhere Flexibilität könnte man mit dem Einsatz von XSLT erreichen.

Hier ein Beispiel für ein StyleSheet:

Die Einträge vor den geschweiften Klammern entsprechen den im XML-Dokument eingeführten Tags (dort in der <tag>- und </tag>-Notation).

In den Klammern folgen Anweisungen wie Elemente dieser Art dargestellt werden sollen.

```
dic { }  
eintrag { display:block; padding:5px; }  
stichwort { font-weight:bold; }  
cat { color:green; font-style:italic; margin-left:10px; }  
trans { display:block; font-style:italic; }  
dekl { margin-left:10px; display:block; color:#BBBBBB; font-style:italic; }  
satz { margin-left:10px; font-size:small; display:block; }  
trans2 { margin-left:10px; display:block; }  
t2_stichwort { font-weight:bold; }  
t2_trans { }  
satz_eintrag {font-weight:bold;}  
satz_trans { display:block; }  
verweis { margin-left:10px; color:#AAEE00; }
```

3.5 Beispiel für einen Eintrag

Der Eintrag zum Stichwort "wyratuk" XML-Repräsentation beispielsweise so aus:

```

<dic>
  <eintrag>
    ... anderer Eintrag
  </eintrag>

  <eintrag>

    <stichwort>wyratuk</stichwort>

    <verweis>s. songo</verweis>

    <trans>Knollen in der Asche backen</trans>

    <satz>
      <satz_eintrag>Svbvrv vyrag-nytyhyn.</satz_eintrag>
      <satz_trans>Back mir die Bataten.</satz_trans>
    </satz>

    <satz>
      <satz_eintrag>Hom ulmv wyraho-felvg fobik jahaltuk.</satz_eintrag>
      <satz_trans> Man legt den Hom zum Backen in die Asche, nachher (wenn er gar ist) nimmt
man ihn heraus.</satz_trans>
    </satz>

    <trans2>
      <t2_stichwort>wyraho-feruk</t2_stichwort>
      <t2_translation>zum Backen in die Asche legen</t2_translation>
    </trans2>

    <trans2>
      <t2_stichwort>komuk vyratuk</t2_stichwort>
      <t2_translation>Knollen unter Steinen halbgar backen</t2_translation>
    </trans2>

    <cat/>

    <dekl>wyrag-taruk</dekl>
    <dekl>-naptuk</dekl>
    <dekl>-nutuk</dekl>
    <dekl>-haruk</dekl>

  </eintrag>

</eintrag>
  ... anderer Eintrag
</eintrag>
</dic>

```

Bindet man das oben angegebene Stylesheet ein und lässt sie sich in einem handelsüblichen Webbrowser darstellen erhält man folgende Darstellung:

wyratuk s. **songo**

Knollen in der Asche backen

Svbvrv wyrag-nytynyhyn.

Back mir die Bataten.

Hom ulmv wyraho-felvg fobik jahaltuk.

Man legt den Hom zum Backen in die Asche, nachher (wenn er gar ist) nimmt man ihn heraus.

wyraho-feruk zum Backen in die Asche legen

komuk wyratuk Knollen unter Steinen halbgar backen

wyrag-taruk

-naptuk

-nutuk

-haruk

4. Vorgehen

In diesem Kapitel wird der Vorgang der Umformung und Überführung der Daten in das neue Format beschrieben.

4.1 Extrahieren der semi-strukturierten Daten

Da es sich um eine große Datenmenge (14 000 Zeilen mit 71 000 Wörtern) handelt, wäre eine manuelle Bearbeitung sehr Zeitaufwendig. Da die Einträge aber einem groben logischen Aufbau folgen, ist es möglich zumindest einen Teil der Daten automatisch zu extrahieren.

Im Fall des Angorruk-Wörterbuch geschieht das mithilfe der Skriptsprache Ruby. Die verwendeten Programme finden sich im Anhang.

4.2 Probleme

4.2.1 Uneinheitlicher Aufbau der Einträge

Bis auf Stichwort und Übersetzung sind alle Teile eines Eintrags optional; Phrasen lassen sich schwer von Übersetzungen unterscheiden, Deklinationsformen werden auf unterschiedlichste Arten angegeben, als Vollformen, mit elliptischer Anfang, mit Bindestrichen gefolgt von Formstücken oder Mischformen.

Kommentare werden ohne spezielle Kennzeichnung an beliebiger Stelle eingefügt: Am Anfang, am Ende, manchmal im laufenden Text. Sogar ein menschlicher Betrachter kann oft nicht entscheiden ob es sich um einen Zusatz zum Beispielsatz / zur Beispielphrase handelt, oder einer Anmerkung zum ganzen Eintrag, oder zu einem speziellen Wort der Phrase gehört.

5. Neue Möglichkeiten

Durch das Trennen von Darstellung und Logik erreicht man höhere Flexibilität der Daten. Es ist nun sehr leicht möglich die Daten maschinell zu verarbeiten. Für die daraus resultierenden Möglichkeiten möchte ich hier ein ein paar Beispiele geben.

5.1 Ein Wörterbuch

Mit Hilfe von XML und CSS lassen sich einfach vielgestaltige Wörterbücher erstellen:

Bsp. 1:

agan

Baum in Zusammensetzungen mit besonderen Eigenschaften

E tu vsa agan.

Das ist ein Tabubaum.

E tu anggen uruk agan.

Dieser Baum trägt Früchte.

(n)

aganuk

neuer Trieb, Ableger

Haly ysalvwag jago-ferikim aganuk lagaptuk.

Wenn man Bananen pflanzt, sprießen Ableger daraus hervor.

Bsp. 2:

agan

*Baum in Zusammensetzungen mit besonde-
ren Eigenschaften*

E tu vsa agan.

Das ist ein Tabubaum.

E tu anggen uruk agan.

Dieser Baum trägt Früchte.

(n)

aganuk

neuer Trieb, Ableger

Haly ysalvwag jago-ferikim aganuk lagaptuk.

Wenn man Bananen pflanzt, sprießen Ableger daraus hervor.

5.2 Ein Korpus

Aus der Xml-Datei lässt sich mit Hilfe von Textextraktionstechniken eine Liste von Sätzen extrahieren. (Auszug)

```
Anden-angge ( an-angge ) hyrag-tarusa.
Ik Idenburgh angge-reg.
Ik nuk-ogo lyt hom ynggyla angejagon wak laruk.
Kvbag nanggelem-togo embygy, jamy-rymygyn.
Aben ap unusurukmv enenggelem-atusa.
angelem-ane-ruruk.
Hyjap svbvrv jagaluk-ogo lyt enenggema laruk.
Ap anggema nogolvg wagrusa.
Nebe anggin-atikik.
Malik ebe anggin teberisi.
Nvjvg angginen horog horog laruk lagy.
Anggolowam hyrag-tarisi vlv hat anggolowam-teg uruk.
Angguruken lagalvg Sengfeng anggolopma ap in-atuk.
Ap Fungfung anggolopma lvgat-atvg pileam wamburuk.
Sabal ynaben ap sawijon anggul-eneptuk.
Hele holtuk lahy-angge famen anggvngga eneg holyhy.
Ap selijon men tog fanowap-tirikim wam anggyrang naruk.
Hyjap malik yndag-tarikim ysyngan anvm paltuk.
Ap mi mi lvhaltuk.
Ap jvnggvluk-oho lyt lvhaloho laruk.
An sum yno wak lag vlv mon-nabehek.
```

5.3 Ein primitives Übersetzungsprogramm

Mit wenigen Handgriffen lässt sich daraus ein Programm generieren um einfache Sätze zu übersetzen. (siehe Anhang)

```
mac-andi:~/Desktop/Ethnologie Hauptsemair andi$ ruby tools/trans.rb ang.xml

Wörterbuch eingelesen, bitte Satz eingeben:

Wam asimagvn unduk fano.
|Schwein| asimagvn |Geschmack| lgut, schön|.

Pulema laruk lyt naluk angginen turuk lagy.
```

pulema lgehen, fließen, strömen, wehen... contr. kommen l lyt naluk angginen lmachenl lagy.

Ap ari ebe along.

ap ldas dortl lKörper, Personl lreich, einflussreich, bedeutend, angesehenl.

5.4 Aufspüren von Lücken im Wörterbuch mit Hilfe eines Korpus

Man kann testen, ob alle in den Beispielsätzen verwendeten Wörter auch im Lexikon enthalten sind. Eine erste Überprüfung zeigt z.B. , dass die vier am häufigsten verwendeten Wörter nicht im Lexikon enthalten sind.

```
andismac:~/time ruby look_for_missing_words.rb ang.xml korpus.txt
```

```
Beginne Wörterbuch einzulesen
```

```
Wörterbuch eingelesen
```

```
Bearbeite Beispielsätze
```

```
Beispielsätze bearbeitet
```

```
Bereite Ausgabe nicht gefundener Wörter vor
```

```
373 ap
333 aben
221 lyt
139 uruk
120 toho
99 teg
90 hyjaben
80 og
73 roho
66 an
63 naruk
56 felvg
54 wituk
52 reg
48 maliken
37 ynaben
37 ysaruk
36 seni
35 ynap
35 wan
34 eneptuk
34 ruruk
33 ano
32 weregma
32 osit
31 war
30 laha
30 latusa
30 ouk
```

6. Fazit

Die Digitalisierung vorhandener Daten erweist sich als aufwendig. Es gibt kein "Patentrezept" da die Datensätze zu unterschiedlich sind. Jeder Ersteller und Bearbeiter benutzt seine ihm vertrauten Programme und entwickelt seine eigenen Konventionen.

Dies macht eine automatische Digitalisierung unmöglich und eine Nachkorrektur durch einen menschlichen Bearbeiter ist unverzichtbar. Dennoch lässt sich durch den Einsatz maschineller Extraktionsmethoden viel Zeit sparen.

Trotz des hohen Aufwands ist eine Digitalisierung und damit auch Normierung alter Daten sinnvoll. Verzichtet man auf diesen Schritt riskiert man dass aufwendig gewonnene Forschungsdaten für Nachfolgenden Bearbeiter unauffindbar wenn nicht gar unnutzbar werden.

Literatur

Austin, Peter K.: Data and language Documentation in Gippert, Himelmannm, Mosel (2006):
Essentials of Language Documentation. Berlin: Walter de Gruyter

W3C, Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C (2006)

Ziegler, Friedrich: Das Wörterbuch in der Arbeit des Ethnologen - am Beispiel der Angurruk-
Sprache

Anhang

I. x_ang.rb

Dieses Programm extrahiert aus dem in eine reine Textdatei umgewandelten Angorruk-Wörterbuch Daten und gibt diese als ein strukturiertes XML Dokument aus.

```
#!/usr/bin/ruby -w

#==Autor: Andreas Neumann
#==Datum: 10.02.2008
#==Zusammenfassung: Extrahiert Daten aus dem Wörterbuch der Angguruk-Sprache,
# benutzt REXML um XML Dokumente zu erstellen

#== Aufruf: ruby x_ang.rb Wörterbuch-Textdatei

$KCODE="u"

class AnggurukExtractor

# Objekt initialisieren, startet einlesen
  def initialize(q)
    #Wörterbuch anlegen
    @dic=Hash.new()

    #Reguläre Ausdrücke zum Extrahieren
    @entry_form='[-\w ,!()=]+'\
    @gloss='[-\w ,.;+()]+\
    @cat=Regexp.new('\([^\)]+\)'), 'U')

    @reg=Regexp.new("(#{@entry_form})\t+(#{@gloss})", 'U')

    @vergleiche=Regexp.new('\b(?:vgl|s)\. ', 'U')

    einlesen(q)
  end

  def extract_rest(e,h)
    e.each do linfol
      info.chomp!()
      case info
      # vgl. s.
      when /^s+$/
        next
      when @vergleiche
        h[:verweis]=info
        #Enthält die Zeile ein Satzzeichen am Ende handelt es sich um einen Satz
        when /[.!?]\)?$/
          unless h[:satz] :
            h[:satz]=Array.new()
          end
          h[:satz] << info
          #Kurzes Beispiel mit Übersetzung
          when /^s*[\^t]+\t+\w+[\w\s() ,.]+$/
            unless h[:trans2] :
              h[:trans2]=Array.new()
            end
          end
        end
      end
    end
  end
end
```

```

        h[:trans2] << info
    when /\w* ?- ?\w+/ then
        #Deklination
        unless h[:dekl] :
            h[:dekl]=Array.new()
        end
        h[:dekl] << info
    else
        # STDERR.puts info unless info=~ /<EINTRAG\/>/
    end
end
end

# Liest Datei ein, als Trenner wird <EINTRAG/> erwartet
def einlesen(quelldatei)
    File.open(quelldatei).each("<EINTRAG/>") do |eintrag|
        parse(eintrag.chomp)
    end
end

#Versucht Form des Eintrags zu erkennen
def parse(eintrag)
    #Eintrag klassifizieren
    #Hat der Eintrag mehrere Untereinträge? TYP 2 -> aufspalten zu TYP1
    if eintrag =~ /\^d/
        eintrag.split(/\^d+\)\s*/).each do |untereintrag|
            add_to_dic(untereintrag)
        end
    #Normaler Eintrag
    else
        add_to_dic(eintrag)
    end
end

# Stichwort mit Übersetzung
def add_info(stichwort,trans,rest)

    cat=check_for_cat(stichwort)
    unless @dic[stichwort] :
        @dic[stichwort]=Array.new()
    end

    h=Hash.new()
    h[:stichwort]=stichwort.gsub(/\s+|\s+$/, "") if stichwort
    h[:cat]=cat
    h[:trans]=trans

    extract_rest(rest,h)

    @dic[stichwort] << h

end

# Trägt Daten gemäß der inneren Struktur in das Wörterbuch ein
def add_to_dic(eintrag)
    e=eintrag.split(/\$/ )
    #In der ersten Zeile steht der Obereintrag mit Übersetzung
    case e[0]
        when /\d+\)/ && @reg then
            add_info($1,$2,e[1..-1])
        when /nur in/ then
            e[1] =~ @reg
            add_info($1,$2,e[2..-1])
        else
            #STDERR.puts e[0]
    end
end

```

```

end

# Prüft ob eine Kategorieangabe vorhanden ist und extrahiert diese
def check_for_cat(s)
  if s =~ @cat
    cat=$1
    s.gsub!(cat,"")
    return cat
  else
    return nil
  end
end

#Gibt das Wörterbuch als XML aus
def print_xml
  require "rexml/document"
  f=File.new("ang.xml","w")
  doc=REXML::Document.new()
  doc << REXML::XMLDecl.new(1.0,"utf8")
  #doc << '<?xml-stylesheet type="text/css" href="ang.css" ?>'
  doc.add_element(REXML::Element.new("dic"))
  @dic.delete(nil) #???
  @dic.keys.sort!.each do |key|
    @dic[key].each do |e|
      eintrag=REXML::Element.new("eintrag")

      e.each_key do |art|
        # Hier wird der Inhalt gesplittet
        case art

          when :dekl || :verweis then
            e[art].each do |eintrag_or_dekl|
              x=eintrag.add_element(art.to_s)
              x.text=eintrag_or_dekl
            end

          when :trans2 then
            e[art].each do |y|
              x=eintrag.add_element(art.to_s)

              a,b=y.scan(/[\^\t]+/)

              w=x.add_element('t2_stichwort')
              w.text=a

              v=x.add_element('t2_translation')
              v.text=b
            end

          when :satz then
            i=1
            while e[art][i]:
              x=eintrag.add_element(:satz.to_s)
              satz_eintrag=x.add_element('satz_eintrag')
              satz_eintrag.text=e[art][i-1]
              satz_trans=x.add_element('satz_trans')
              satz_trans.text=e[art][i]
              i+=2
            end

          else
            x=eintrag.add_element(art.to_s)
            x.text=e[art]
          end
        end
      end
    end
  end
end

```

```

        doc.root.add_element(eintrag)
      end

    end

    doc.write(f,0)

  end

end

#####

Vers2=AnggurukExtractor.new(ARGV[0])
Vers2.print_xml()

```

2. trans.rb

Ein einfaches Übersetzungsprogramm. Nach dem Starten erwartet es die Eingabe eines Satzes. Jede Eingabe wird mit "Return/Enter" abgeschickt.

```

#!/usr/bin/ruby -w

===Autor: Andreas Neumann
===Synopsis: Sehr einfaches Übersetzungsprogramm durch Ersetzung
#
===Aufruf: ruby trans.rb Wörterbuch

require "rexml/document"

# Datei öffnen
f=File.new(ARGV[0])
# Hash für Übersetzung
h=Hash.new()

# Als XML-Dokument behandeln
doc=REXML::Document.new f
#Einlesen
doc.elements.each("*/eintrag") do |x|

h[x.elements["stichwort"].to_s.gsub(/<[^>+>/,"").gsub(/<[^&]+&/,"")] = x.elements["trans"].to_s.gsub(/<[^>+>/,"|")
end
# Nach Wortlänge sortieren, damit Phrasen Einzelwörtern vorgezogen werden
wb=h.keys.sort {|a,b| a.length <=> b.length}

puts "Beenden mit STRG+D"
puts "Wörterbuch eingelesen, bitte Satz eingeben:"

while input=$stdin.gets
  input.downcase!
  wb.each do |k|
    input.gsub!(/\\b#{Regexp.quote(k)}\\b/, "#{h[k]}")
  end

  puts input
end

```

3. look_for_missing_words.rb

Dieses Programm hilft Lücken im Wörterbuch ausfindig zu machen. Beim Aufruf wird als erstes Argument das Wörterbuch und als zweites der Korpus übergeben.

Das Programm gibt die Wörter die nicht erkannt wurden frequenzgeordnet aus.

```
#!/usr/bin/ruby -w

#==Autor: Andreas Neumann
#==Synopsis: Versucht lücken im Wörterbuch zu finden, (vorsicht lange Laufzeit)

require "rexml/document"

dic=File.open(ARGV[0])
korpus=File.open(ARGV[1])
bekannt=Array.new()
nicht_erkannt=Hash.new()

# Als XML-Dokument behandeln
doc=REXML::Document.new dic
#Einlesen
puts "Beginne Wörterbuch einzulesen"
doc.elements.each("*/eintrag") do |x|
    b e k a n n t < <
    x.elements["stichwort"].to_s.gsub(/<[^>+>/,"").gsub(/^(^)*\)/,"").gsub(/[\(\)]/, "")
end
bekannt.sort!
puts "Wörterbuch eingelesen"

#Liest Beispielsätze ein
puts "Bearbeite Beispielsätze"
korpus.each do |zeile|
    zeile.downcase!
    #Entfernt bekannt Wörter aus Beispielsätzen
    bekannt.each do |wort|
        zeile.gsub!(/b#{wort}\b/, "")
    end
    # Alles was übrig bleibt wurde nicht erkannt und wird gesammelt
    zeile.scan(/\w+/).each do |nicht_erkanntes_wort|
        nicht_erkannt[nicht_erkanntes_wort] ||= 1
        nicht_erkannt[nicht_erkanntes_wort] += 1
    end
end
puts "Beispielsätze bearbeitet"

puts "Bereite Ausgabe nicht gefundener Wörter vor"
# Nicht erkannte Wörter nach Häufigkeit sortieren und Ausgeben
k=nicht_erkannt.keys()
k.sort {|a,b|nicht_erkannt[b]<=>nicht_erkannt[a]} .each do |wort|
    puts "#{nicht_erkannt[wort]}\t#{wort}"
end
```