3 Verarbeitung von symbolischen Daten

VORTRAG

3.1 Einführung

Unter symbolischer Information verstehen wir im engeren Sinne Daten, die sich auf Signale beziehen. Als einfachstes Beispiel könnte man eine Verschriftung einer Aufnahme als symbolische Information bezeichnen. Man nennt solche Informationen 'symbolisch' (manchmal auch 'kategorial'), um sie von den physikalischen Signalen zu unterscheiden. Wir können folgende Tabelle aufmachen, die den Unterschied etwas verdeutlicht:

	Physikalisches Signal	Symbolische Information
Repräsentation	Abtastwerte (Zahlen)	Zeichenketten
Speicherbedarf	Hoch	Niedrig
Informationstyp	Empirisch	Kategorial (meistens)
Beispiele	Sprachsignal	Transliteration
	Laryngo-Signal	Segmentierung
	F0-Verlauf	Akzente
	Formantverläufe	Formantpositionen/-breiten

Andere Beispiele für symbolische Information:

Transliteration:	Liste der Wörter plus Marker für andere Ereignisse, z.B. Geräusche, linguistische Effekte, Hesitationen, Pausen, Zögerungen, etc.
Transkript:	Liste der Phoneme bzw. Allophone einer Äußerung
Segmentierung:	Liste von Teilstücken einer Äußerung, den Segmenten
Phonetische Segmentierung:	Liste der Allophone mit segmentaler Information
Silbensegmentierung:	dito mit Silben statt Allophonen
Wortsegmentierung:	dito mit Wortgrenzen
Dialogakt-Segmentierung:	Abbildung der Wortkette auf *Dialogakte*
Prosodische Segmentierung:	Position und Klasse des prosodischen Ereignisses
Syntaxbaum:	Syntaktische Struktur einer Äußerung

Als *Meta-Informationen* bezeichnen wir zusätzliche wichtige Informationen zu einer Äußerung, die sich aber nicht direkt auf das Signal beziehen: Sprecher: Name, Geschlecht, Alter, Herkunft, Ausbildung, Größe, Raucher/Nichtraucher, Dialekt, Dialekt der Eltern, Grundschule, Beruf, Wohnort Aufnahmebedingungen: Mikrophon, Aufnahmetechnik, Raum, Akustik, Geräuschquellen, Task, Art der Sprache (spontan/gelesen etc)

Für den Phonetiker, der mit sehr großen Datenmengen experimentiert, ist es essentiell, solche symbolischen Informationen automatisch verarbeiten zu können. Z.B. um bestimmte Klassen von Signalen aus einem Korpus zu sammeln. Daher lernen wir im Folgenden einige wichtige UNIX-Tools zur Manipulation von symbolischer Information kennen.

GEMEINSAME ÜBUNG

3.2 Beispiele für symbolische Daten

Kopieren Sie die Dateien (mit allen Unterverzeichnissen!) in `/homes/ekurs2/SS10/Symbolic-Data` in Ihr Arbeitsverzeichnis und versuchen Sie deren Art einzuordnen in:

- Sprecher-Information
- Aufnahmebedingungen
- Symbolische Information zu einer Aufnahme

```
w026_pk.trl
w026_pk.trp
g121a/*.par
w026_pk.rpr
ABD.spr
```

ÜBUNG:
Lesen Sie das folgende Kapitel und beantworten Sie dann Fragen des Dozenten (ev. als Hausaufgabe)

3.3 Zeilenbasierte symbolische Daten

3.3.1 Beispiel Partitur-Datei

In der Phonetik ist das physikalische Signal fast immer ein Sprachsignal. Zu dem Signal kann es aber eine praktisch unbegrenzte Anzahl verschiedener symbolischer Beschreibungsformen geben. Um diese geordnet bearbeiten und darstellen zu können, stellen wir uns vor, dass diese in einer Art Musik-Partitur als verschiedene Stimmen über das gleiche Grundthema angeordnet sind. Daher der Name *BAS Partitur Format*.

Die einzelnen Stimmen der der Partitur nennen wir *Spuren* oder englisch *tiers* (gebräuchlich sind auch *layers*, *levels*, usf.). Alle haben irgendeinen Bezug zum zugrundeliegenden Sprachsignal, können also zeitlich *synchronisiert* werden. (Genau wie auch die Noten einer Partitur synchron übereinander angeordnet sind!). Darüber hinaus gibt es aber noch weitere Möglichkeiten, die einzelnen tiers zu verbinden, und zwar über sog. Wort-Links oder *symbolische Links*. Ein symbolischer Link ist ganz einfach eine Zahl, die uns sagt, welchem 'Wort' das beschriebene Ereignis zugeordnet werden soll. Mit diesen beiden einfachen Mechanismen können nun alle möglichen Arten von symbolischer Information geordnet beschrieben werden.

(Skizze : Beispiel einer Partitur)

3.3.2 Wie sieht eine solche Partitur-Repräsentation als Datei aus?

Möglichst einfach. Ein *Partiturfile* besteht aus einer 7bit ASCII-Datei, die auf jedem Rechner zu lesen und zu bearbeiten ist (Plattform-Unabhängigkeit). Jede Zeile beginnt mit einem *Label*, dass uns sagt, welche Information aus dieser Zeile gelesen werden kann (man sagt auch: jedes Label definiert eine Syntax und eine Semantik der folgenden Zeile). Zum Beispiel

```
SPN: KAP
```

sagt uns diese Zeile, dass der Sprecher das Kürzel `'KAP'` hatte. Die Label sind natürlich einheitlich festgelegt und können in der Dokumentation [1] nachgeschaut werden.

Der erste Teil des Files ist ein genormter *Header*, d.h. ein Teil der uns allgemeine Informationen (Meta-Daten) über das Sprachsignal gibt. Z.B.

- Name des zugehörigen Signalfiles
- Abtastrate
- Anzahl der Bytes pro Abtastwert
- Bitauflösung
- Byteorder (01 oder 10)

Aber auch Dinge wie:

- Name oder Kürzel des Sprechers
- Datum der Aufnahme
- Name des Korpus etc.

Der Header-Teil beginnt immer mit dem Label `'LHD'` und ended mit dem Label `'LBD'`.

Beispiel für einen einfachen Header:

```
LHD: Partitur 1.2.3
REP: Muenchen
SNB: 2
SAM: 16000
SBF: 01
SSB: 16
NCH: 1
SPN: KAP
LBD:
```

Die erste Zeile (LHD) sagt uns die verwendete Version des Partitur Formats. Die Zeile `'REP'` definiert den 'place of recording', in diesem Falle 'Muenchen'. Die Zeile `'SNB'` sagt uns, dass pro Abtastwert 2 Bytes verwendet wurden. Die Zeile `'SAM'` definiert die 'sampling rate', die Abtastrate, zu 16kHz. Die Zeile `'SBF'` gibt die Reihenfolge der Bytes an: 01 Die Zeile `'SSB'` gibt die Bitauflösung wieder, hier 16 Bit. `'NCH'` schließlich ist die Anzahl der in diesem File gespeicherten Kanäle (1) und `'LBD'` schliesst den Header-Teil ab.

Nach diesem Header-Teil, der ein Minimum an Informationen enthalten muss, folgen beliebig viele *Spur-Blöcke*, die die einzelnen tiers der Partitur beschreiben. Wieder hat jeder dieser Blöcke sein eigenenes Label und seine eigene Syntax und Semantik. Die Reihenfolge der einzelnen Blöcke spielt keine Rolle.

Beispiel:

Die folgenden Zeilen definieren eine sog. `ORT` tier (orthographische Spur), d.h. die nackten gesprochenen Wörter:

[1] www.phonetik.uni-muenchen.de/Bas/BasFormatsdeu.html

```
ORT: 0  tsch"u"s
ORT: 1  denn
```

Diese Äußerung enthält also nur zwei Wörter. Wörter sind im Partitur Format so definiert, dass sie alle semantischen Einheiten bezeichnen, die vom Artikulationstrakt des Sprechers stammen. Geräusche ohne semantischen Gehalt (wie Atmen, Husten,etc.) werden nicht als Wörter gezählt. Ein Grenzfall sind Häsitationen wie 'ähm' oder 'hm'; diese werden als Wörter gezählt. Wie man sieht, enthält dieser Block überhaupt keine Informationen darüber, wo denn nun die beiden Worter im physikalischen Sprachsignal liegen. Es handelt sich hier um einen 'Klasse 1' tier, das ist ein tier, der nur rein kategoriale Elemente enthält, ohne irgendeinen direkten Bezug zur Zeitachse. Andere Klasse 1 tiers sind z.B.:

Transliteration

```
TR2: 0 <#Klicken> tsch"u"s <#Klicken>
TR2: 1 denn .
```

Kanonische Aussprache

```
KAN: -1 <nib>
KAN: 0  tS'y:s
KAN: -1 <nib>
KAN: 1  dEn
```

Was für andere Klassen gibt es?

Es gibt insgesamt nur 5 verschiedene Grundklassen. Klasse 1 haben wir oben bereits kennengelernt. Die anderen 4 Klassen sind:

Klasse 2	: Spuren mit zeitlicher Relation, zeitkonsumierend
Klasse 3	: Spuren mit zeitlicher Relation, nicht zeitkonsumierend
Klasse 4	: Spuren mit zeitlicher Relation und symbolischer Relation, zeitkonsumierend
Klasse 5	: Spuren mit zeitlicher Relation und symbolischer Relation, nicht zeitkonsumierend

Beispiele:

Phonetische Segmentation (Klasse 4)

```
MAU: 0 2879 -1 <p:>
MAU: 2880 639 -1 <nib>
MAU: 3520 479 0 t
MAU: 4000 959 0 S
MAU: 4960 1919 0 y:
MAU: 6880 2879 0 s
MAU: 9760 639 -1 <nib>
MAU: 10400 1119 -1 <p:>
```

Prosodische Labelung (Grenzen und Akzente; Klasse 5)

```
PRB:    6532    -1      TON: H*; FUN: PA
PRB:    10083   -1      BRE: B3; TON: ?%
```

Hier ist ein Beispiel für eine (kurze) komplette Partitur Datei:

```
LHD: Partitur 1.2.3
REP: Muenchen
SNB: 2
SAM: 16000
SBF: 01
SSB: 16
NCH: 1
SPN: KAP
LBD:
TRL: 0 <#Klicken>
TRL: 0 tsch"u"s
TRL: 0 <#Klicken> .
KAN: -1 <nib>
KAN: 0 tS'y:s
KAN: -1 <nib>
DAS: 0 @(BYE BA)
PRB:  6532  -1 TON: H*; FUN: PA
PRB:  10083  -1 BRE: B3; TON: ?%
WOR:  0  2615  -1 <#>
WOR:  2616  1249  -1 <#Klicken>
WOR:  3866  6217  -1 tsch"u"s
WOR:  10084  1349  -1 <#>
WOR:  11434  291  -1 <P>
MAU: 0 2879 -1 <p:>
MAU: 2880 639 -1 <nib>
MAU: 3520 479 0 t
MAU: 4000 959 0 S
MAU: 4960 1919 0 y:
MAU: 6880 2879 0 s
MAU: 9760 639 -1 <nib>
MAU: 10400 1119 -1 <p:>
TR2: 0 <#Klicken> tsch"u"s . <#Klicken>
ORT: 0 tsch"u"s
```

((Fragen zu Partitur-Dateien))

GEMEINSAME ÜBUNG

3.4 Der Suchbefehl grep

Der UNIX-Befehl `grep` dient zum Suchen *in* Dateien (i.G. zum Befehl `find` oder `locate`, die zum Suchen *von* Dateien dienen!)

Im Directory `g121a` liegt eine Sammlung von sog. Partitur-Files (Endung *.par). Eine solche Sammlung nennen wir (zusammen mit den Signalen!) *Korpus*. Wir wollen zunächst feststellen, wie oft die Hesitation `'<"ah>'` in diesem Korpus vorkommen. Der Befehl `grep` sucht zeilenweise in Dateien nach bestimmten Mustern. Der allgemeine Aufruf ist dabei:

```
cip1 % grep <Muster> file1 [file2 file3 ...]
```

Alle Zeilen, in denen das Muster vorkommt, werden zusammen mit dem Filenamen ausgegeben. Z.B.

```
cip1 % grep '<"ah>' g121a/*.par
```

gibt alle Zeilen aus, in denen der String `'<"ah>'` vorkommt.

Nun fällt sofort auf, dass jedes `'<"ah>'` mehrfach vorkommt, weil dieses Label (eine Markierung) nicht nur in einer *Spur* (einer Beschreibungsebene) vorkommt. Wir können das vermeiden, indem wir uns vorher mit einem weiteren 'grep' auf die Zeilen einer einzigen Spur beschränken, z.B. auf die Spur 'ORT':

```
cip1 % grep '^ORT:.*<"ah>' g121a/*.par
```

Das ist der gleiche Befehl wie oben, aber das Muster ist komplizierter geworden. Es enthält sog. *Meta-Zeichen* mit besonderer Bedeutung, wie sie in *Regulären Ausdrücken* (Mustern) vorkommen.

VORTRAG

3.5 Muster: Reguläre Ausdrücke

Ein regulärer Ausdruck ist ein *Muster* für eine Gruppe von Strings. Dazu werden sogenannte Metazeichen verwendet, die nicht das Zeichen bedeuten, sondern eine besondere Bedeutung haben:

`^`	: Zeilenanfang
$	: Zeilenende
`.`	: beliebiges Zeichen
`*`	: beliebige Wiederhohlung des vorangegangenen Zeichen (auch 0-mal!)
`[...]`	: genau ein Zeichen der Menge ...
`[^...]`	: genau ein Zeichen NICHT aus der Menge ...

Es ist üblich einen regulären Ausdruck zwischen Schrägstriche zu setzen, um anzuzeigen, dass es sich um ein Muster handelt (z.B. `/Flo.*/`).

Beispiele:

Muster	Mögliche Strings
`/Flo.*/`	Flo, Flori Florian, aber nicht: flori
`/g???c/`	g000c, gabcc, gAAAc, aber nicht g0c
`/Nr [1-3[0-9]/`	Nr 10, Nr 23, Nr 39, aber nicht Nr 100, Nr 54

((Mehr Beispiele an der Tafel))

GEMEINSAME ÜBUNG

Das Muster `'^ORT:.*<"ah>'` bedeutet also im Klartext:
'Suche die Zeilen, die am Zeilenanfang mit 'ORT:' beginnen, dann eine beliebig lange Kette von beliebigen Zeichen (auch 0) und dann den String `'<"ah>'` enthalten.'

(Die einfachen Hochkommata vor und nach dem regulären Ausdruck sind notwendig, weil hier im Ausdruck ein `'"'` und ein `'*'` vorkommt. Würde man die Hochkommata weglassen, würde die Shell meinen, man gibt ihr einen Ausdruck mit Joker (*) und versuchen, dazu passende Filenamen zu finden.)

Nun haben wir alle Zeilen auf dem Bildschirm, aber eigentlich wollten wir nur wissen, wieviele es sind. Also zählen wir in einer Pipeline einfach die Zeilen:

```
cip1 % grep '^ORT:.*<"ah>' g121a/*.par | wc -l
```

Zählen Sie auf die gleiche Weise auch die Hesitationen `'<"ahm>'`, `'<hm>'` und `'<h"as>'` und addieren Sie alle zusammen. Dann zählen wir, wieviele Äußerungen der Korpus überhaupt enthält:

```
cip1 % ls g121a/*.par | wc -l
```

Wenn wir die beiden Zahlen untereinander dividieren, bekommen wir die mittlere Anzahl von Hesitationen `'<"ah>'` pro Äußerung in diesem Korpus.

GEMEINSAME ÜBUNG

3.6 Arbeiten mit 'gawk'

'Gawk' ist ein Programm, das genau wie 'grep' Files zeilenweise nach Mustern (regulären Ausdrücken) absucht und für jedes Muster eine bestimmte Aktion durchführen kann. Der simpelste Aufruf von 'gawk' ist

```
cip1 % gawk <Programm> <Input-File>
```

Ein GAWK-Programm besteht immer aus folgenden Kommandos:

```
<Muster1> { Aktion1 }
<Muster2> { Aktion2 }
...
```

'gawk' sucht dann in jeder Zeile des Files nach den Mustern und wenn es eines davon findet, wird die entsprechende Aktion dahinter ausgeführt.

Zum Beispiel gibt folgender Befehl

```
cip1 % gawk '/^ORT/ { print $3 }' g121axx0_000_OLV.par
```

nur die dritte Spalte (`$3`) aus allen Zeilen, die mit 'ORT' beginnen auf den Bildschirm (standard output).

Wenn man das Muster weglässt, wird jede Zeile verarbeitet:

```
cip1 % gawk '{ print $3 }' g121axx0_000_OLV.par
```

Bei längeren Programmen ist es handlicher, diese nicht immer wieder einzutippen, sondern in einem File zu speichern. Außerdem will man fast immer viele Files auf einmal in einer Pipe verarbeiten. Der Aufruf sieht dann so aus:

```
cip1 % cat file1 file2 file3 ... | gawk -f <Programm-File>
```

ÜBUNG

Schreiben Sie mit einem Editor (z.B. pico) folgendes Programm in das File hesi.awk (achten Sie darauf in Ihrem Gruppen-Dir zu sein!):

```
BEGIN { count = 0 }
/^ORT:.*<"ah>/ {
          count = count + 1
        }
/^ORT:.*<"ahm>/ {
          count = count + 1
        }
/^ORT:.*<hm>/ {
          count = count + 1
        }
END   { print "Anzahl der Hesitationen: " count }
```

Rufen Sie es auf mit dem Befehl

```
cip1 % cat ../Partitur/* | gawk -f hesi.awk
```

Genau wie andere Skriptsprachen kennt gawk Variablen und Kontroll-Kommandos (if-Verzweigung, Schleifen, etc.). Eine vollständige Beschreibung der Sprache ist in der Manual-Page von gawk zu finden:

```
cip1 % man gawk
```

Erweitern Sie Ihr Programm-File so, dass es außerdem noch die Anzahl der Wörter insgesamt ausgibt (zweiter Zähler, Muster für ORT-Zeilen).

Erweitern Sie Ihr Programm-File so, dass es außerdem noch die relative Anzahl von Hesitationen pro Wort ausgibt.

Testen Sie nun alle Sprecher (w.o.), Münchner Sprecher (`cat ../Partitur/m???d*`), Karlsruher Sprecher (`cat ../Partitur/n???k*`), Bonner Sprecher (`cat ../Partitur/m???n*`) und Kieler Sprecher (`cat ../Partitur/j???a*`)

Wer verwendet im Mittel mehr Hesitationen?

GEMEINSAME ÜBUNG

3.7 Automatisches Editieren mit 'sed'

Wir haben schon in den ersten Stunden sog. Filter kennengelernt, die Operationen auf Text-Files automatisch ausführen können. Bestes Beispiel für ein Filterprogramm ist der UNIX-Befehl 'tr' (translate), der einzelne Zeichen vertauschen, zusammenfassen ('squeezen') oder löschen kann.

```
cip1 % echo "Mein Geld" | tr 'M' 'D'
```

Manchmal reicht die einfache Funktionalität von 'tr' nicht mehr aus, z.B. wenn man längere Zeichenketten ersetzen möchte. Nehmen wir an, wir haben einen Fehler in unseren Partitur-Files entdeckt. Und zwar hat jemand konsequent `'n"ahmlich'` statt `'n"amlich'` geschrieben. Wir können das z.B. für die Münchner Files feststellen mit dem Befehl:

```
cip1 % grep 'n"ahmlich' ../Partitur/m???d*
```

Wir können jetzt nicht einfach mit 'tr' alle 'h'-Zeichen löschen; also brauchen wir einen komplexeren Mechanismus, der sämtliche Editierfunktionen beherrscht. Der *stream editor* 'sed' ist so ein komplexes Filter. Genauso wie 'tr' ließt er von standard input, filtert das File und schreibt es wieder nach standard output.

```
cip1 % echo 'n"ahmlich'
cip1 % echo 'n"ahmlich' | sed 's/n"ahmlich/n"amlich/'
```

Der allgemeine Aufruf von 'sed' ist sehr ähnlich wie bei 'gawk':

```
cip1 % cat file | sed <Programm>
```

Wie bei awk kann man das Programm auch in eine Datei `prog.sed` schreiben und dann den folgenden Aufruf verwenden:

```
cip1 % cat file | sed -f prog.sed
```

Das 'Programm' bei 'sed' besteht nur aus einzelnen Editierbefehlen. Von allen Befehlen der wichtigste (und am häufigsten gebrauchte) ist der 'substitute' Befehl:

```
s/muster/replacement/
```

Also wird in obigem Beispiel der String `'n"ahmlich'` durch den String `'n"amlich'` ersetzt. Ein kleines angehängtes 'g' ('global') bewirkt, dass alle vorkommenden Muster in einer Zeile ersetzt werden (sonst nur das erste auftretende Muster):

```
s/muster/replacement/g
```

Der Muster-Teil des 'substitute' Befehls ist wie bei 'gawk' oder 'grep' ein regulärer Ausdruck. Nochmal zur Wiederholung:

`^` :	Zeilenanfang
`$` :	Zeilenende
`.` :	ein beliebiges Zeichen
`*` :	eine beliebige Wiederholung
`[xyz]` :	Eines von 'x' oder 'y' oder 'z'
`[a-g]` :	Ein Zeichen von 'a' bis 'g'
`[^xyz]` :	Ein Zeichen, das nicht 'x' oder 'y' oder 'z' ist
`\( \)` :	Musterklammerung (kann im Replacement-Teil mit `'\1'` wieder eingesetzt werden)

ÜBUNG

Kopieren Sie sich das fehlerhafte File in Ihr Gruppen-Dir. Ersetzen Sie mit 'sed' in dem fehlerhaften File alle falschen `'nähmlich'` durch `'nämlich'`.

ÜBUNG

4 Übung zu *sed* und *awk*

4.1 Aufgabe

Stellen Sie sich vor, die Steuerfahndung steht vor der Tür und sie haben auf Ihrem PC noch sämtliche gefälschten Bilanzen gespeichert. Einfach Löschen geht nicht, weil die Beamten dann misstrauisch werden. Sie müssen ganz schnell handeln, weil Sie die Steuerfahnder schon auf der Treppe hören.

1. Entwerfen Sie ein sed Programm, das im eingegebenen Text sämtliche vorkommende EUR-Beträge bis 999 EUR durch 'XXX' ersetzt, also z.B.

   ```
   EUR 445,78   ->   EUR XXX,78
   EUR 87,00    ->   EUR XXX,00   (beachten Sie, dass das Komma noch an der richtigen Stelle ste
   ```

   Andere Zahlenangaben sollen aber erhalten bleiben, z.B.

   ```
   '4. Vorstandssitzung'  ->  '4. Vorstandssitzung'
   ```

   Testen Sie Ihren Befehl mit der Datei 'Bilanzen' in Ihren Gruppendirectory.

```
% cat Bilanzen | sed 'PROGRAM' | less
```

2. Sind Beträge mit verschieden langen Lücken zwischen 'EUR' und dem Betrag korrekt abgebildet worden? Z.B.

   ```
   'EUR       45,78'   ->   'EUR XXX,78'
   ```

   Wenn nicht, erweitern Sie den Befehl dazu, dass diese auch erfasst werden. Könnten Sie Ihren Befehl so modifizieren, dass die Länge der Lücke genau gleich bleibt?

3. Jetzt sollen auch Tausender und Millionenbeträge (auf die kommt es ja an!) alle korrekt abgebildet werden, z.B.

   ```
   'EUR  1.900.000,-  ->  EUR XXX,-
   ```

4. Das folgende macht zwar keinen unmittelbaren Sinn, wenn die Steuerfahndung kommt, aber es ist eine schöne Übung: Schreiben Sie ein gawk Programm, das im Inputfile nach allen EUR-Beträgen sucht und diese korrekt zusammenaddiert und schließlich die Summe ausgibt.

   Tip: Schreiben Sie Ihr gawk Programm nicht mehr auf die Kommandozeile (wie in den Beispielen oben), sondern in ein Programm-File. Der Aufruf ändert sich dann zu:

   ```
   % cat Bilanzen | gawk -f PROGRAMM-FILE
   ```

   Auf diese Weise können sie leichter mehrzeilige Programme schreiben und müssen nicht dauernd auf der Kommandozeile editieren.

   Gehen Sie schrittweise vor:

   - Schreiben Sie zunächst ein Programm, das alle Zeilen mit EUR Beträgen ausgibt (Denken auch mal an grep und eine Pipe-Struktur)
   - Erweitern Sie es, damit nur die EUR Beträge ausgegeben werden (die einzelnen FELDER einer Zeile werden in awk durch die Variablen $1, $2, ... repräsentiert)
   - Schauen sie sich in der Man Page zu gawk (`% man gawk`) die String-Funktionen gsub() an und überlegen, wie Sie die EUR-Beträge in ganze Zahlen umformatieren können. Geben Sie sie zur Kontrolle wieder aus.
   - Schließlich addieren Sie sie auf und geben das Endergebnis aus.

5. Irgendein Computer-Oldtimer überedet Sie, Ihre Bilanz in LaTeX umzuformatieren. Unter anderem müssen dazu alle Umlaute anders geschrieben werden:

   ```
   ä -> "a
   Ü -> "u
   ö -> "o
   ß -> "s
   Ä -> "A
   Ü -> "U
   Ö -> "O
   ```

   Z.B.

   ```
   'Ötzi ißt Übel.'  ->  '"Otzi i"st "übel'
   ```

Schreiben Sie ein sed Programm `latex.sed`, das diese Übersetzung automatisch macht und testen Sie es mit Ihrer Bilanz:

```
cip1 % cat Bilanzen | sed -f latex.sed
```

6. Schreiben Sie ein gawk Programm, das im Input-Text sämtliche Zahlen (nur Kardinalzahlen) in ausgeschriebene Zahlwörter umwandelt. Also z.B.:

   ```
   'Ich esse 46 Bananen.'  ->  'Ich esse sechsundvierzig Bananen.'
   ```

4.2 Lösungen

1. `cat Bilanzen | sed 's/EUR [0-9]*,/EUR XXX,/' | less`

2. `cat Bilanzen | sed 's/EUR *[0-9]*,/EUR XXX,/' | less`
   `cat Bilanzen | sed 's/\(EUR *\)[0-9]*,/\1XXX,/' | less`

3. `cat Bilanzen | sed 's/\(EUR *\)[0-9.]*,/\1EUR XXX,/' | less`

4. —

5. Siehe File `~schiel/bin/utf82latex.sed`
   Test z.B. mit:
   `cat Bilanzen | sed -f ~schiel/bin/utf82latex.sed`

6. Siehe File `/share/local/lib/digit2words.awk`.
   Test z.B. mit:
   `echo '3456545' | gawk -f /share/local/lib/digit2words.awk`