

Grundlegendes: Software, Datenstrukturen, Encoding und das geheime Leben meines Computers

Inhalt

Was lernen wir in diesem Tutorial?	2
Einführung	2
Software	2
1. Texteditoren	2
2. Tabellenkalkulationsprogramme: Excel und Calc	3
3. Datenauswertung und Statistik: R und RStudio	8
Praktische Übung	10
Datenstrukturen	10
CSV- und TSV-Dateien	10
Dateinamenerweiterungen ändern	11
Encoding	11
Die Universalgrammatik der Suchanfrage: Reguläre Ausdrücke	12
Die wichtigsten regulären Ausdrücke	14
Boolesches "oder"	14
Gruppierung durch Klammern	14
Wildcard (Platzhalter)	14
Quantifikation	14
Anfang und Ende	15
Escape Strings	15
Klammerausdrücke	15
Wo ist was auf der Tastatur?	15
Lookaround Assertions	16
Geiz ist geil: Wie man reguläre Ausdrücke „non-greedy“ macht	16
Übungsaufgaben	18
Literatur	18

Was lernen wir in diesem Tutorial?

- Wir lernen Programme kennen, die für die Arbeit mit Daten, insbesondere mit historischen Korpusdaten, nützlich sind.
- Wir erfahren, wie man Dateien mit diesen Programmen öffnet und wie man bei Bedarf ihre Dateinamenerweiterung ändern kann.
- Wir lernen geeignete Dateiformate für die Speicherung tabellarischer Daten wie z.B. Korpuskonkordanzen kennen.
- Wir erfahren (in Grundzügen), wie Zeichenkodierung funktioniert und warum unterschiedliche Kodierungen (z.B. UTF-8 vs. ANSI) zu viel Frustration führen können – v.a. auf Windows-Rechnern.
- Wir lernen reguläre Ausdrücke kennen, mit deren Hilfe wir in Korpora, Texterarbeitungsprogrammen und anderer Software effizient suchen können.

Einführung

Jedes Forschungsprojekt ist anders, aber ganz grundsätzlich lassen sich alle empirischen Projekte ganz grob auf zwei Arbeitsschritte herunterbrechen: Datengewinnung und Datenanalyse. In der germanistischen Sprachwissenschaft gewinnen wir Daten z.B. aus Korpora oder aus Fragebogenstudien und Experimenten, aber es sind auch andere Datenquellen denkbar (z.B. Namenregister und Telefonbücher in onomastischen Projekten). Empirische Linguistik ist also immer auch „Data Science“, um ein modernes Schlagwort zu verwenden. Daher ist es wichtig, über Grundkenntnisse im Bereich der Datenverarbeitung zu verfügen – übrigens nicht nur in der Linguistik, sondern mittlerweile eigentlich in fast allen Lebensbereichen.

In der Korpuslinguistik ist Datenverarbeitung in aller Regel Textverarbeitung. Bei Textverarbeitung denken viele von Ihnen wahrscheinlich an Programme wie Word. Für den Umgang mit Korpusdaten brauchen wir aber andere Werkzeuge, die wir im Folgenden kennenlernen werden. Erfreulicherweise stehen die meisten der Programme, die wir in diesem und den weiteren Tutorials kennenlernen werden, kostenlos und teilweise auch quelloffen zur Verfügung. Eine Ausnahme ist Microsoft Excel, das ich in diesen Tutorials deshalb mit diskutiere, weil viele LeserInnen damit bereits vertraut sein dürften; zudem haben viele Hochschulen Office-Lizenzen. Allerdings bietet sich gerade für historische Korpusdaten auch die kostenlose Alternative Calc von LibreOffice an.

Software

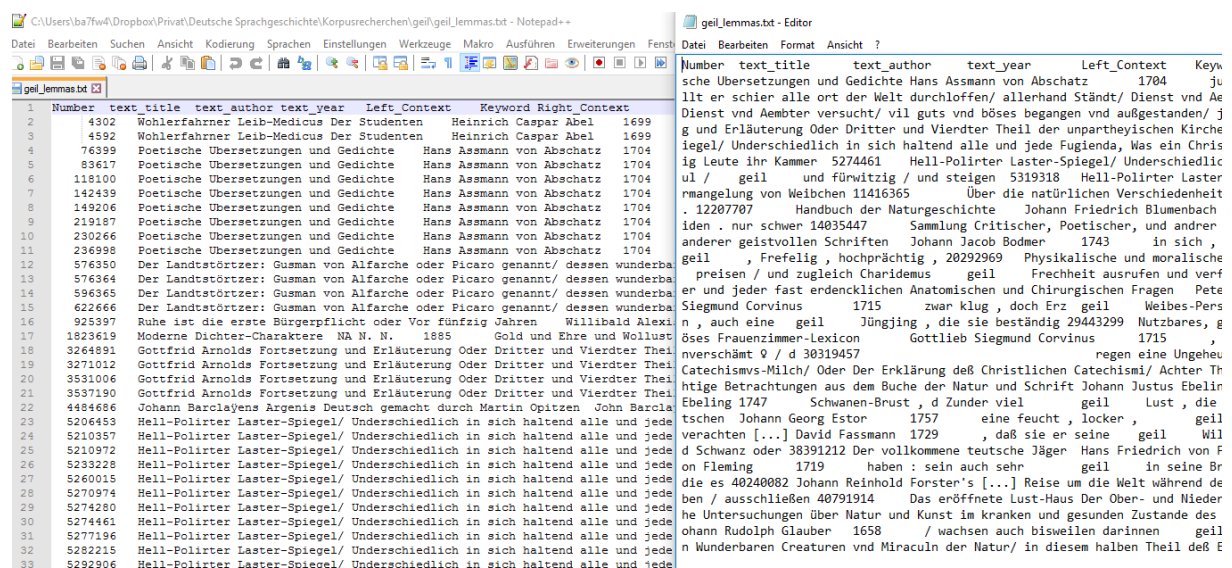
1. Texteditoren

Um mit Korpusdateien und -konkordanzen zu arbeiten, können sich Texteditoren als äußerst hilfreich erweisen. Auf den meisten Betriebssystemen sind schon einfache Editoren vorinstalliert, die aber meist unseren Zwecken nicht genügen – so kann z.B. der vorinstallierte Windows-Editor mit vielen csv-Dateien (s.u.) nur bedingt umgehen, und auch der Funktionsumfang der „Suchen&Ersetzen“-Funktion ist stark eingeschränkt. Daher empfehle ich, auf jeden Fall einen der folgenden Editoren zu installieren:

- für Windows: **Notepad++**. Kann viel, kostet nichts und hat nur den Nachteil, dass er (bisher) lediglich für Windows verfügbar ist.

- für Mac: z.B. **TextWrangler**. Kann ähnlich viel wie Notepad++ und ist ebenfalls kostenlos, hat aber den Nachteil, dass man manchmal beim Start gefragt wird, ob man nicht auf die kostenpflichtige Variante BBEdit upgraden möchte.
- für Linux: z.B. **Vim Editor** oder **Notepadqq**

Plattformübergreifend ist auch der Open-Source-Editor **Atom** verfügbar, der allerdings eher für fortgeschrittene BenutzerInnen zu empfehlen ist (und außerdem derzeit noch ein paar kleinere Bugs hat). Auch unterstützt die Suchen-und-Ersetzen-Funktion nicht alle sog. Lookaround Assertions (siehe Unterkapitel [Lookaround Assertions](#) im Abschnitt „Die wichtigsten regulären Ausdrücke“ am Ende dieses Tutorials), die man zwar de facto im Texteditor nur selten braucht, die aber trotzdem sehr nützlich sein können. Auch der kostenlose Editor **VisualStudio Code** von Microsoft ist eher für Fortgeschrittene und eher zum Erstellen von Code als für die Verarbeitung von Text geeignet.



Number	text_title	text_author	text_year	Left_Context	Keyword	Right_Context
1	4302	Wohlerfahner Leib-Medicus Der Studenten	Heinrich Caspar Abel	1699		
2	4592	Wohlerfahner Leib-Medicus Der Studenten	Heinrich Caspar Abel	1699		
3	76399	Poetische Übersetzungen und Gedichte	Hans Asmann von Abschatz	1704		
4	83617	Poetische Übersetzungen und Gedichte	Hans Asmann von Abschatz	1704		
5	119100	Poetische Übersetzungen und Gedichte	Hans Asmann von Abschatz	1704		
6	142439	Poetische Übersetzungen und Gedichte	Hans Asmann von Abschatz	1704		
7	149206	Poetische Übersetzungen und Gedichte	Hans Asmann von Abschatz	1704		
8	219187	Poetische Übersetzungen und Gedichte	Hans Asmann von Abschatz	1704		
9	230266	Poetische Übersetzungen und Gedichte	Hans Asmann von Abschatz	1704		
10	236998	Poetische Übersetzungen und Gedichte	Hans Asmann von Abschatz	1704		
11	576350	Der Landstörtzer: Gusman von Alfarche oder Picaro genannt/ dessen wunderba				
12	576364	Der Landstörtzer: Gusman von Alfarche oder Picaro genannt/ dessen wunderba				
13	596365	Der Landstörtzer: Gusman von Alfarche oder Picaro genannt/ dessen wunderba				
14	622666	Der Landstörtzer: Gusman von Alfarche oder Picaro genannt/ dessen wunderba				
15	925397	Ruhe ist die erste Bürgerpflicht oder Vor fünfzig Jahren Willibald Alexi				
16	1823619	Moderne Dichter-Charaktere NA N. N. 1885	Gold und Ehre und Wollust			
17	3264891	Gottfrid Arnolds Fortsetzung und Erläuterung Oder Dritter und Vierdter Thei				
18	3271012	Gottfrid Arnolds Fortsetzung und Erläuterung Oder Dritter und Vierdter Thei				
19	3531006	Gottfrid Arnolds Fortsetzung und Erläuterung Oder Dritter und Vierdter Thei				
20	3537190	Gottfrid Arnolds Fortsetzung und Erläuterung Oder Dritter und Vierdter Thei				
21	4484686	Johann Barclayens Argenis Deutsch gemacht durch Martin Opitzzen John Barcia				
22	5206453	Heil-Polirter Laster-Spiegel/ Unterschiedlich in sich haltend alle und jede				
23	5210357	Heil-Polirter Laster-Spiegel/ Unterschiedlich in sich haltend alle und jede				
24	5210972	Heil-Polirter Laster-Spiegel/ Unterschiedlich in sich haltend alle und jede				
25	5233228	Heil-Polirter Laster-Spiegel/ Unterschiedlich in sich haltend alle und jede				
26	5260015	Heil-Polirter Laster-Spiegel/ Unterschiedlich in sich haltend alle und jede				
27	5270974	Heil-Polirter Laster-Spiegel/ Unterschiedlich in sich haltend alle und jede				
28	5274280	Heil-Polirter Laster-Spiegel/ Unterschiedlich in sich haltend alle und jede				
29	5274461	Heil-Polirter Laster-Spiegel/ Unterschiedlich in sich haltend alle und jede				
30	5277196	Heil-Polirter Laster-Spiegel/ Unterschiedlich in sich haltend alle und jede				
31	5282215	Heil-Polirter Laster-Spiegel/ Unterschiedlich in sich haltend alle und jede				
32	5292906	Heil-Polirter Laster-Spiegel/ Unterschiedlich in sich haltend alle und jede				

2. Tabellenkalkulationsprogramme: Excel und Calc

Die Daten, mit denen wir in der empirischen Linguistik arbeiten, kommen häufig in tabellarischem Format, z.B. Korpus-Konkordanzen oder die Ergebnisse von Fragebogenstudien. Oft wollen wir diese Tabellen dann noch mit weiteren Daten anreichern. Angenommen beispielsweise, wir haben eine Tabelle mit Genitivkonstruktionen und möchten untersuchen, welche Faktoren die Voran- oder Nachstellung des Genitivs beeinflussen. In diesem Fall könnten wir z.B. eine Annotationsspalte hinzufügen, in der wir die Belebtheit des Kopfnomens annotieren, und eine weitere, in der wir annotieren, ob es sich um eine appellativische Personenbezeichnung oder einen Eigennamen handelt.

Es gibt nun verschiedene Möglichkeiten, das zu tun. Meine allererste Korpusrecherche vor vielen Jahren (für eine Hausarbeit zur *ung*-Nominalisierung, was später auch Thema meiner Dissertation werden sollte) lief so: Ich machte eine Tabelle in Word (!), bzw. genauer: eine Tabelle für jeden Korpustext, fügte ein paar Annotationsspalten hinzu und wertete die Annotationen hinterher aus, indem ich sie manuell (!!) auszählte. Das ist ziemlich unnötig und nicht zur Nachahmung empfohlen.

0	Beleg im Kontext	Stelle	Arg.	als pr. Kpl.	mit adj. Mod	Art.	Pl.	Maj.
1.	wir auch nichts weniger vns allen zu Trost/ sterckung deß Glaubens/ vnd zur besserung vnsers Sündhafftigen lebens reichen möge/	NOBD-1620- KT-081	O	(zu)				
2.	wir auch nichts weniger vns allen zu Trost/ sterckung deß Glaubens/ vnd zur besserung vnsers Sündhafftigen lebens reichen möge/	NOBD-1620- KT-081	X	zu+ Art		b (enklit. mit Präp. versch m.)		
3.	Da ist immer Furcht/ Hoffnung / vnnd zu letzt der Todt/	NOBD-1620- KT-081						X
4.	Vnd daß dem also sey/ bezeugen solches neben der täglichen erfahrung /	NOBD-1620- KT-081			X	b		
5.	Es ist auch solch Bethel in der außtheilung deß gelobten Lands/ dem Stam_ BenJa=#min eingereumbt/ vnd zugeeignet worden.	NOBD-1620- KT-081	X	in + Art		b		
6.	Das ist/ *Domus DEI.* Ein Hauß Gottes/ von wegen der herrlichen Offenbahrung so Jacob allda widerfahren.	NOBD-1620- KT-081			X	b		X
7.	Gleich wie Jacobs *in #tent *	NOBD-1620- KT-081	(X)	(zu)				

Fig. 1: Abschreckendes, aber authentisches Beispiel einer Konkordanz in Word.

Stattdessen sollten wir uns an ein echtes Tabellenkalkulationsprogramm wagen. Hier müssen wir allerdings im Hinterkopf behalten, dass Tabellenkalkulationsprogramme zumeist eher auf Zahlen denn auf Texte ausgelegt sind. Dennoch können sowohl Excel als auch Calc, mit denen wir in diesen Tutorials arbeiten werden, relativ gut mit Textdaten umgehen, von ein paar kleineren Schönheitsfehlern abgesehen. Man muss nur wissen, wie man dem Programm, vereinfacht gesagt, beibringt, die Daten richtig zu lesen.

Nehmen wir eine x-beliebige csv-Datei, deren sich viele in diesem Begleitmaterial finden. Wenn Sie Microsoft Office installiert haben, ist die Wahrscheinlichkeit hoch, dass diese Dateien standardmäßig mit Excel verknüpft sind. Das heißt, wenn Sie einfach auf eine der Dateien doppelklicken, dann öffnet sich Excel – und wenn Sie Pech haben, sehen Sie ein ziemliches Durcheinander:

	A	B	C	D	E	F	G	H	I	J	K
1	Source Left Key1 Right Year Decade Lemma										
2	A15 SteinwÄ	NA NA	vorprogrammiert								
3	BVZ15 oder well done - ein Geschmackserlebnis ist vorprogrammiert. Zum Start des neuen Jahres NA NA	vorprogrammiert									
4	HMP15 LESERBRIEFE "Die Altersarmut ist programmiert" Hartz-IV-Fakten Check Die NA NA	programmiert									
5	U15 bestehe die nÄrchsten Katastrophen sind programmiert; schlieÄlich verlaufen durch NA NA	programmiert									
6	U15 die ihre ist vorprogrammiert." Die Kreditinstitute NA NA	vorprogrammiert									
7	P15 Zeit." Konflikte mit der Regierung sind programmiert. Immerhin sollen auch die NA NA	programmiert									
8	A15 Jahr ins Parlament kommt. Konflikte sind programmiert. Bachmann: Das wird sich NA NA	programmiert									
9	BVZ15 oder well done - ein Geschmackserlebnis ist vorprogrammiert. Zum Start des neuen Jahres NA NA	vorprogrammiert									
10	NZZ15 im kommenden Herbst. Die Verwirrung ist programmiert. WÄhlen ist eine diffizile NA NA	programmiert									
11	T15 stellen si der nÄrchste Ausfall ist vorprogrammiert. Mario Matt selbst gibt NA NA	vorprogrammiert									
12	SBL15 Ä«Eine Billag-Steuer von 1000 Franken ist programmiertÄ» Hans-Ulrich Bigler NA NA	programmiert									
13	M15 Partei. Das Debakel bei der OB-Wahl ist vorprogrammiert (ich tippe mal auf einen NA NA	vorprogrammiert									
14	P15 Die nÄrchste Verfassungswidrigkeit ist programmiert Fortpflanzungsmedizin. Nach NA NA	programmiert									
15	P15 "Die nÄ von Stephan NA NA	programmiert									
16	NON15 HÄrchtleistungen an. Der Erfolg ist vorprogrammiert. NA NA	vorprogrammiert									
17	HMP15 am Flughafen Lange Wartezeiten sind programmiert: In Hamburg und Stuttgart NA NA	programmiert									
18	U15 niedrige Briefe an NA NA	programmiert									
19	HMP15 um il denn hinter Lenas RÄcken NA NA	programmiert									
20	NON15 den unterschiedlichen Interessengruppen sind programmiert." Der Hohenberger Johann NA NA	programmiert									
21	NUN15 verzÄlgert sich NÄrchster Engpass ist programmiert Die Äberschrift zeugt von der NA NA	programmiert									
22	NUZ15 stÄrzten gestern regelrecht ab. Streit ist programmiert. Denn Tsipras und sein NA NA	programmiert									
23	NON15 es be ist vorprogr NA NA	vorprogrammiert									
24	NON15 City-Dirtrun in Amstetten. Änderungen sind vorprogrammiert Bei der Neuauflage im NA NA	vorprogrammiert									
25	NON15 City-Dirtrun in Amstetten. Änderungen sind vorprogrammiert Bei der Neuauflage im NA NA	vorprogrammiert									
26	M15 Dienststellenleiter werden - der Streit ist programmiert. "Vorgesetzter wird man durch NA NA	programmiert									
27	NUN15 AussI prophezeit das 28-jÄhrige NA NA	programmiert									
28	A15 zu Äberqueren - ein ordentlicher RÄffel ist vorprogrammiert. Dieser ist aber gepaart mit NA NA	vorprogrammiert									
29	PRF15 die IÄber um ihr Wild. Konflikte sind programmiert. Der oberÄsterreichische NA NA	programmiert									

Fig. 2: Eine in Excel geöffnete CSV-Datei.

Sie ahnen wahrscheinlich schon, dass die Datei so nicht aussehen sollte – schließlich wollen wir eine Tabelle und keinen wilden Wörterwust mit kryptischen Sonderzeichen. Dass die Datei so angezeigt wird, liegt daran, dass Excel zwei Vorannahmen macht:

1. Es nimmt an, dass die einzelnen Tabellenspalten durch Kommas voneinander abgetrennt sind. Bei der hier geöffneten Datei ist das nicht der Fall: Hier fungieren Tabstopps als Trennzeichen (dazu mehr unten).
2. Es nimmt an, dass die Daten in ANSI kodiert sind; die hier geöffnete Datei ist allerdings in UTF-8 kodiert (dazu ebenfalls mehr unten).

Deshalb empfehle ich, Dateien niemals direkt in Excel zu öffnen. Wenn Sie mit Excel arbeiten wollen, dann wählen Sie besser einen der beiden folgenden Wege:

entweder

1. Rechtsklick auf die Datei, die Sie öffnen wollen
2. „Öffnen mit“ auswählen und einen Texteditor (z.B. Notepad++) auswählen
3. Mit Strg+A den gesamten Text markieren
4. Mit Strg+C den gesamten Text kopieren
5. Excel öffnen und dort mit Strg+V den gesamten Text einfügen
6. Auf den kleinen Button mit Einfüge-Optionen klicken, der in der letzten sichtbaren Spalte erscheint (s. Fig. 3 unten) und dort den **Textimport-Assistenten** auswählen

oder

1. Ein leeres Arbeitsblatt in Excel öffnen
2. Im Reiter „Daten“ die Option „Aus Text“ wählen
3. Die Datei auswählen
4. Nun erscheint ebenfalls der **Textimport-Assistent**

Nun geht es für beide Varianten gleich weiter: Im Textimport-Assistenten...

1. ...kann man im ersten Fenster die Kodierung auswählen (hier: UTF-8) und angeben, ob die Daten trennzeichen-getrennt sind (das sind sie) oder eine feste Spaltenbreite haben (diese Option abwählen). Auf „Weiter“ klicken.
2. Im nächsten Fenster kann man das Trennzeichen auswählen (hier: Tabstopp). Wichtig ist außerdem, im Dropdown-Menü „Textqualifizierer“ (i.d.R.) das leere Feld anzuwählen. Textqualifizierer sind Zeichen, die anzeigen, dass Text zusammengehört, und zwar selbst dann, wenn ein Texttrenner darin vorkommt. Wenn ich also z.B. mit Kommas als Trennzeichen arbeite "und diesen Text, also genau den hier, in Anführungszeichen setze", dann wird *also genau den hier* nicht abgetrennt (also in eine eigene Spalte gesetzt), sondern vielmehr wird die gesamte Passage als zusammengehörig betrachtet. Standardmäßig werden bei Excel Anführungszeichen als Textqualifizierer erkannt. Gerade bei Korpus-Konkordanzen ist es aber oft so, dass wir im Kontext ein öffnendes, aber kein schließendes Anführungszeichen haben (oder umgekehrt), wie z.B. Beleg P15 in Fig. 4 zeigt. Dann wird im schlimmsten Fall alles bis zum nächsten Anführungszeichen, das in irgendeinem Beleg auftritt, als *ein* zusammengehöriger Text behandelt, und uns gehen viele Belege verloren.

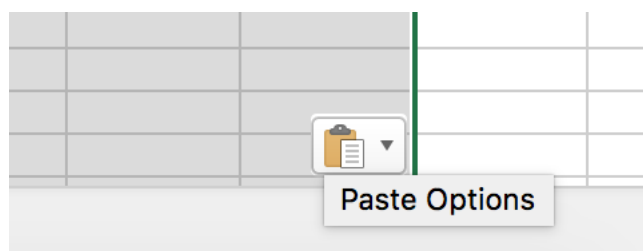


Fig. 3: Der kleine Button mit Einfügeoptionen (siehe Schritt 6)

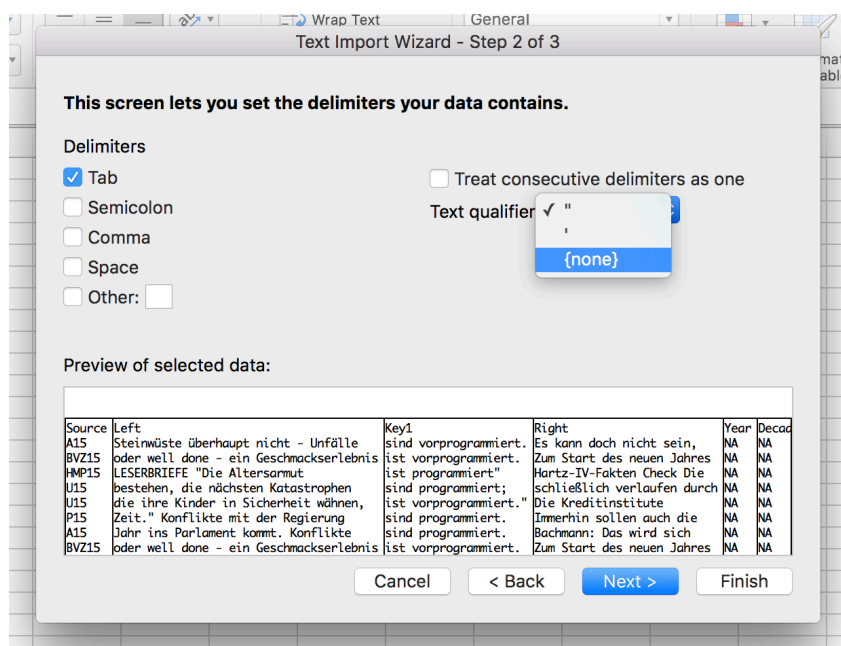


Fig. 4: Menü zur Auswahl von Texttrennern und Textqualifizierern.

Im Idealfall sehen wir die Konkordanz nun so, wie wir sehen wollen. Bei Calc geht das alles etwas einfacher: Wenn wir auf die csv-Datei rechtsklicken und „Öffnen mit > LibreOffice“ auswählen, dann öffnet sich automatisch ein Fenster, das dem Excel-Textimport-Assistenten sehr ähnlich ist. Hier können wir ebenfalls die Kodierung, das Trennzeichen und den Textqualifizierer (in Fig. 5: *text delimiter*) auswählen. Bei Letzterem ist darauf zu achten, dass so etwas wie „{none}“ nicht zur Verfügung steht, sondern im Dropdown-Menü nur einfaches und doppeltes Anführungszeichen zur Auswahl stehen - allerdings kann man einfach in das Feld klicken und durch Klick auf die Löschen-Taste angeben, dass es keinen Textqualifizierer gibt.

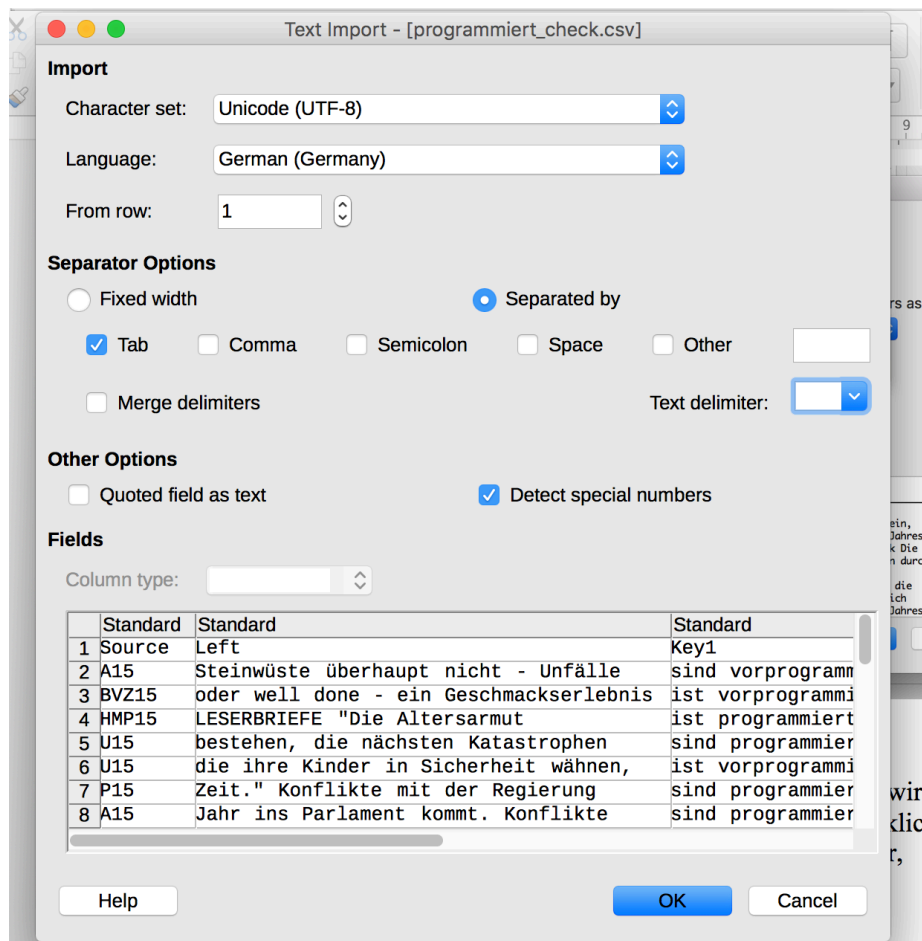


Fig. 5: Textimport-Assistent von LibreOffice Calc.

Praktische Übung

Lesen Sie die Datei *programmiert.csv*, die sich im Begleitmaterial findet (im Ordner *Excel_Einstieg* oder [hier](#)), in Excel oder Calc ein, indem Sie den oben skizzierten Schritten folgen.

3. Datenauswertung und Statistik: R und RStudio

Die Programmiersprache R hat sich in großen Teilen der Linguistik mittlerweile als de-facto-Standard durchgesetzt und wird insbesondere zur statistischen Datenauswertung verwendet; prinzipiell kann man sie aber z.B. auch zur Datengewinnung (etwa durch Webcrawling) einsetzen. R ist extrem flexibel und kann dank vieler Erweiterungen (sog. Pakete) bei Bedarf auch mit großen Daten- bzw. Textmengen umgehen. An dieser Stelle werden wir allerdings noch nicht auf konkrete Nutzungsmöglichkeiten eingehen, sondern uns lediglich kurz mit der Frage beschäftigen, wie man Daten in R einlesen kann.

Wenn Sie noch gar nicht mit R gearbeitet und es noch nicht installiert haben, müssen Sie das zunächst tun – auf r-project.org gibt es Distributionen für jede Plattform, die sich einfach und selbsterklärend installieren lassen. Nun ist R aber ein konsolenbasiertes Programm, das zunächst keine wirkliche grafische Benutzeroberfläche mit sich bringt. Eine solche kann jedoch in manchen Fällen sehr hilfreich sein; daher benutzen wir im Folgenden das kostenlose RStudio, das sich über r-studio.com herunterladen und installieren lässt. Wichtig: RStudio funktioniert nur, wenn R bereits installiert ist; es genügt also nicht, *nur* RStudio zu installieren.

Wenn wir RStudio öffnen, sehen wir ein viergeteiltes Interface. Für uns wichtig sind die beiden Fenster auf der linken Seite. Unten links sehen wir die Konsole, die quasi dem entspricht, was wir auch beim „reinen“ R finden. Oben sehen wir das Skriptfenster. Ein Skript ist eine Textdatei, in der Code gespeichert ist. Das ist extrem nützlich, denn in der Konsole selbst wird der Code, den man eingibt, nicht gespeichert, sondern einfach nur ausgeführt. In einem Skript dagegen können wir Code zunächst schreiben, ggf. auch so lange damit herumexperimentieren, bis er funktioniert, und statt ihn jedesmal neu eingeben zu müssen, können wir ihn einfach speichern. Zudem können wir auch Kommentare hinzufügen, die erklären, was der Code macht. Je komplexer der Code wird, desto wichtiger ist das – nicht nur für andere Menschen, die den Code möglicherweise irgendwann zu Gesicht bekommen, sondern auch für Sie selbst, wenn Sie ihn irgendwann wiederverwenden möchten. (Bei vielen meiner älteren Skripts habe ich das nicht gemacht und bereue es heute sehr.)

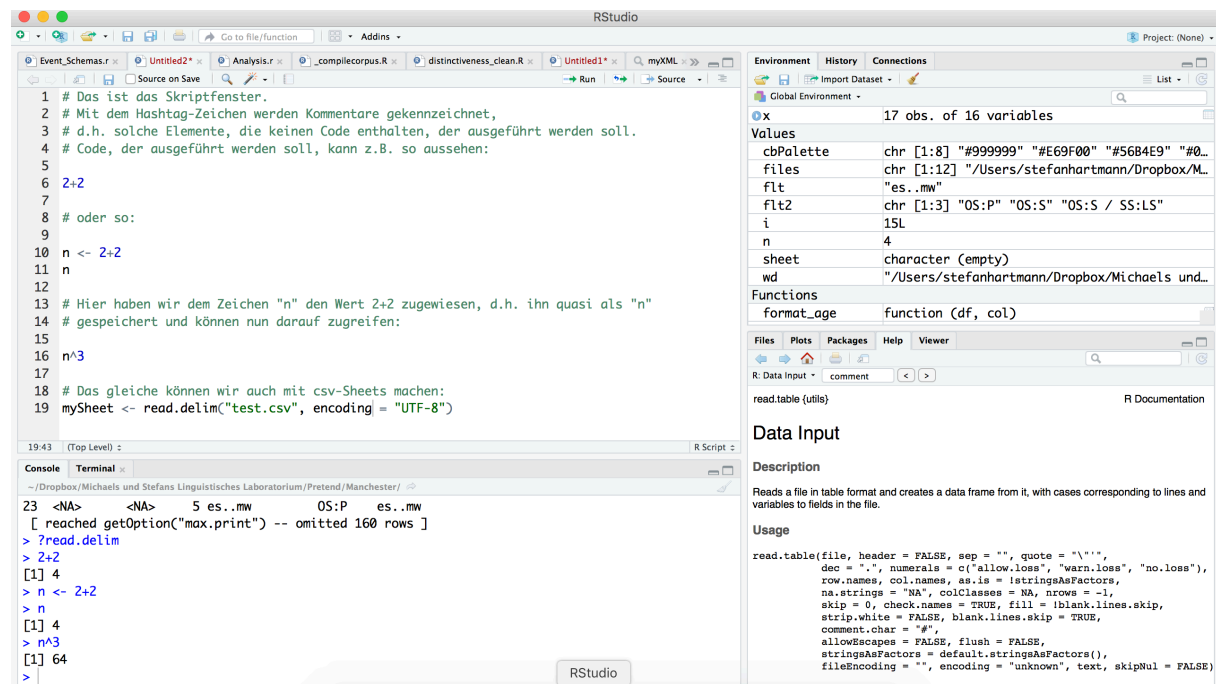


Fig. 6: Interface von RStudio. Oben links: Das Skriptfenster, unten links; Die Konsole; oben rechts: Das Environment; unten rechts: Die Hilfe, die man aufrufen kann, indem man aufrufen kann, indem man einen Funktionsnamen eingibt, dem ein Fragezeichen vorangestellt ist, z.B. `?read.delim`.

R ist eine objektbasierte Programmiersprache, d.h. quasi alles, womit wir arbeiten, ist ein sog. „Objekt“. Wenn wir beispielsweise, wie in Fig. 6 gezeigt, dem Wert $2+2$ (also 4) den Namen n zuweisen, dann ist dieses n ebenfalls ein Objekt. Statt $2+2$ hätten wir auch $sum(2,2)$ benutzen können. sum ist eine **Funktion**; die Angaben in Klammern sind sog. **Argumente**.

In R gibt es fast immer verschiedene Wege zum gleichen Ziel. So gibt es auch verschiedene Möglichkeiten, Dateien einzulesen. Für tab-separierte Dateien empfehle ich:

```
mySheet <- read.delim(file = "test.csv", sep = "\t", quote="", encoding = "UTF-8", stringsAsFactors = FALSE)
```

Die Funktion *read.delim* gehört zu einer Familie von Funktionen zum Einlesen von Daten; stattdessen hätten wir auch die generischere Funktion *read.table* verwenden können. Die Argumente in der obigen Funktion sind:

- a. `file = "test.csv"` – die Datei, die wir einlesen wollen. Hier kann der genaue Dateipfad stehen; liegt die Datei jedoch im derzeitigen Arbeitsverzeichnis (das man mit `getwd()` erfragen und mit `setwd()` setzen kann), dann genügt es, den Dateinamen anzugeben.
- b. `sep = "\t"` gibt an, dass Tabstopps als Trennzeichen benutzt werden. Bei *read.delim* braucht man das eigentlich nicht mit anzugeben, weil Tabstopps standardmäßig als Trennzeichen gesetzt sind. Der Vollständigkeit halber erwähne ich es aber – arbeitet man mit Daten, die andere Trennzeichen haben, z.B. Kommata oder Semikola, muss man sie hier angeben, also z.B. `sep = ";"`.
- c. `quote=""` gibt an, dass keine Zeichen als Textqualifizierer fungieren. Defaultmäßig geht R, wie auch Excel und Calc, davon aus, dass doppelte Anführungszeichen als Textqualifizierer benutzt werden. Deshalb ist es auch hier wichtig, dieses Argument zu setzen, falls man nicht zufällig mit Daten arbeitet, bei denen tatsächlich einmal Anführungszeichen als Textqualifizierer auftreten.
- d. `encoding = "UTF-8"`: Wie bei den Tabellenkalkulationsprogrammen, gilt auch bei R, dass es nicht zwangsläufig von selbst weiß, in welcher Kodierung die Daten vorliegen. Auf Mac- und Linux-Betriebssystemen geht es standardmäßig von UTF-8 aus (dann ist also diese Angabe unnötig, und sie muss nur gemacht werden, wenn die Daten in einem anderen Format vorliegen). Bei Windows hingegen erwartet R Latin-1, da es sich am Default des Betriebssystems orientiert.
- e. `stringsAsFactors = FALSE`: Diese Ergänzung ist für AnfängerInnen am schwierigsten zu erklären. Die kurze Erklärung: Wenn Sie es nicht tun, können schlimme Dinge passieren – ich weiß, wovon ich spreche...

Die etwas längere: Standardmäßig interpretiert R Textstränge als sog. Faktoren. Man kann sich Faktoren ein bisschen wie die Parteienliste auf einem Wahlzettel vorstellen: Es gibt eine endliche Menge an Ausprägungen (Parteien). Sie erscheinen in einer bestimmten Reihenfolge, die aber nicht unbedingt aus den Ausprägungen selber folgt – gäbe es eine Partei „Die ERSTEN“ und eine „Partei des Zweiten Deutschen Fernsehens“, erschienen sie nicht unbedingt als erste und als zweite auf dem Wahlzettel. Auch kann die Liste nicht ergänzt werden: Wenn ich eine nicht zugelassene Partei von Hand dazuschreibe, wird der Wahlzettel ungültig. So ähnlich ist es auch bei Faktoren: Habe ich einen numerischen Vektor, der versehentlich faktorial interpretiert wird (und das ist schon der Fall, wenn sich in der Tabelle versehentlich ein Textelement in eine Spalte mit Zahlen verirrt), kann ich es nicht unmittelbar mit einer Funktion wie *as.numeric()* konvertieren, denn dann ignoriert R die Zahlen und nummeriert stattdessen die Faktoren-Levels durch. Zudem kann ich die entsprechenden Spalten dann nicht verändern (z.B. einzelnen Ausprägungen sinnvollere Namen zuweisen)

oder ergänzen. (Das heißt, genau genommen kann ich es schon, mit der Funktion *as.character()*, aber gleich *stringsAsFactors* auszuschalten, spart uns diesen Schritt).

Daher empfehle ich, immer mit *stringsAsFactors = FALSE* zu arbeiten und bei Bedarf einzelne Vektoren bzw. Spalten von Dataframes mit *as.factor()* manuell zu Faktoren zu konvertieren, falls es denn tatsächlich einmal nötig und sinnvoll ist. Wenn man viele Dateien in einer Sitzung einlesen möchte, kann man auch einfach zu Beginn mit *options(stringsAsFactors = FALSE)* die Interpretation von Strings als Faktoren generell ausschalten und muss es nicht bei jedem Einlesen von Daten neu spezifizieren.

Praktische Übung

Lesen Sie die Datei *programmiert.csv*, die sich im Begleitmaterial findet (im Ordner *R_Einstieg*), in R ein, indem Sie den oben skizzierten Schritten folgen. Sie können zum Beispiel folgenden Code verwenden, um die Datei zunächst einzulesen und anschließend den eingelesenen Dataframe zu inspizieren:

```
programmiert <- read.delim(file = "programmiert.csv", sep = "\t",
  quote="", encoding = "UTF-8", stringsAsFactors = FALSE)
View(programmiert)
```

Beachten Sie, dass die Unterscheidung zwischen Groß- und Kleinschreibung bei R wichtig ist: Wenn Sie z.B. *view* anstelle von *View* verwenden, funktioniert der Code nicht; ebenso bei *Read.delim* anstelle von *read.delim*!

Datenstrukturen

CSV- und TSV-Dateien

Für eine problemlose Datenauswertung ist es wichtig, die Daten, mit denen man arbeitet, in einem geeigneten Format zu speichern. Eine Excel-Tabelle z.B. hat den Nachteil, dass das proprietäre .xls(x)-Format, in dem die Tabellen per Default gespeichert werden, nicht unbedingt mit jedem anderen Programm vollständig kompatibel ist. Demgegenüber haben einfache komma- oder tabstopp-separierte Textdateien den Vorteil, dass sie extrem einfach und elegant sind und in Programmen wie R (aber eben auch Excel) problemlos eingelesen werden können. Mehrfach habe ich bereits das CSV-Format erwähnt. CSV steht für *comma-separated values*. Wie der Name sagt, handelt es sich um eine Datei, in der die einzelnen Werte (Tabellenspalten) durch Trennzeichen wie z.B. Kommata abgetrennt sind. Neben CSV gibt es auch TSV (für *tab-separated values*). Verwirrenderweise stößt man relativ oft auf Dateien mit der Endung .csv, die aber mit Tabs statt Kommata als Trennzeichen arbeiten.¹ Linguistisch gesehen ist das ein schönes Beispiel für metonymische Übertragung, wobei die Einfachheit des Formats dafür sorgt, dass diese Übertragung nicht nur metonymisch, sondern eben auch ganz praktisch funktioniert. Denn letzten Endes sind .csv- und .tsv-Dateien einfach nur Textdateien, die mit der Dateiendung .txt genauso funktionieren würden. Und statt Tabstopps oder Kommata kann man z.B. auch &-Zeichen, %-Zeichen usw. als Trennzeichen verwenden.

In Excel gibt es die Möglichkeit, Excel-Tabellen als .csv-Dateien zu speichern, allerdings kann man dort nicht manuell das Trennzeichen festlegen; vielmehr benutzt es automatisch das Semikolon. Dass es das Semikolon benutzt und nicht das Komma, liegt vermutlich daran, dass in der „deutschen“ Notation das Komma als Dezimaltrennzeichen fungiert (z.B. 0,005 - null

¹ Übrigens auch in den vorliegenden Begleitmaterialien...

komma null null fünf); im englischen Sprachraum (und übrigens auch in R!) erfüllt diese Funktion der Punkt.²

Wenn man mit Textdaten arbeitet, muss man das jedoch wiederum im Hinterkopf behalten, denn Texte enthalten ja gelegentlich Semikola. Will man also eine Excel-Tabelle mit Textdaten als .csv speichern, muss man entweder zunächst alle Semikola suchen und ersetzen (durch einen Platzhalter oder gar nichts, d.h. sie entfernen) oder aber eine der anderen Exportvarianten benutzen, die Excel bietet, z.B. tab-separierte .txt-Datei. Wie bereits gesagt, ist eine tab-separierte .txt-Datei quasi nichts anderes als eine .csv- oder .tsv-Datei, und wenn wir das wollen, können wir die so erstellte .txt-Datei hinterher auch einfach zu .csv umbenennen (wie das geht, sehen wir im nächsten Abschnitt) und haben Excel ein Schnippchen geschlagen. Bei Calc müssen wir diesen Umweg nicht gehen, sondern können das Trennzeichen beim Speichern direkt auswählen.

Dateinamenerweiterungen ändern

Manchmal kann es sinnvoll sein, Dateinamenerweiterungen zu ändern: Zum Beispiel könnte es vorkommen, dass man eine Datei mit tab-separierten Werten hat, die als .txt-Datei gespeichert ist und die man in .tsv umbenennen möchte. Unter Windows kann man so auch zügig ein neues R-Skript in einem bestimmten Ordner anlegen: Einfach Rechtsklick > Neu > Textdatei und dann die .txt-Endung durch .R ersetzen.

Allerdings sind Betriebssysteme häufig so eingestellt, dass sie die Dateinamenerweiterungen standardmäßig nicht anzeigen. Unter Windows 10 (und m.W. auch anderen neueren Windows-Versionen) geht das im Windows Explorer – also dem Dateimanager, in dem Sie durch die Dateien auf Ihrer Festplatte navigieren. Dort finden Sie die entsprechende Einstellung unter Datei > Optionen, dort im Reiter „Ansicht“ den Haken bei „Erweiterungen bei bekannten Dateitypen ausblenden“ entfernen. Unter Mac kann man diese Einstellung im Finder (so heißt dort der Dateimanager, in dem man durch die Dateien auf der Festplatte navigiert) vornehmen. Unter Einstellungen > Erweitert kann man hier ein Häkchen setzen bei „Alle Dateinamenerweiterungen anzeigen“.

Encoding

Eine der häufigsten Problemquellen insbesondere in der Benutzung von Excel und R liegt darin, dass die Zeichenkodierung, die die Programme annehmen, nicht mit der Zeichenkodierung des eigentlichen Dokuments übereinstimmen.

Was ist mit Zeichenkodierung gemeint? Ganz einfach: Wenn ich am PC einen Text schreibe (z.B. diesen hier), dann geschieht das letzten Endes, wie Sie wissen, quasi über Einsen und Nullen. Für Buchstaben, Zahlen und andere Zeichen gibt es mehrere Wege, sie sozusagen in Einsen und Nullen zu fassen. Im Folgenden werden wir uns mit zwei Kodierungen näher befassen, von denen es jeweils ein paar einander relativ ähnliche Varianten gibt:

- Das von Windows und Microsoft-Produkten verwendete Windows-1252 ist eine Erweiterung von ASCII (American Standard Code for Information Interchange), dem Standard des American National Standardization Institute (ANSI) – deshalb werden (verwirrenderweise) beide Kodierungen gelegentlich auch als ANSI bezeichnet. Vereinfachend werde ich im Folgenden einfach „ASCII“ für all diese Kodierungen verwenden.
- Auf Linux- und Unix-(Mac-)Betriebssystemen hingegen wird standardmäßig UTF-8 verwendet, das alle Unicode-Zeichen darstellen kann; Kodierungsvarianten, die auf

² Wenn man mit Excel *und* R arbeitet, ist das übrigens generell ein Punkt, den man im Hinterkopf behalten sollte. Ich selbst habe die Voreinstellungen auf das internationale System umgestellt, also so geändert, dass immer der Punkt als Dezimaltrennzeichen fungiert.

Unicode basieren (neben UTF-8 gibt es z.B. auch UTF-16 oder UTF-32) werden manchmal auch selbst als „Unicode“ bezeichnet. Unicode selbst ist ein Standard zur Darstellung von Text, der stetig um neue Zeichen (z.B. auch Smileys und Emoticons) erweitert wird.

Den Varianten ist gemeinsam, dass die Kodierung über sog. Oktette erfolgt, also über Codes, die aus 8 Bits (Informationseinheiten) bestehen. Bei ASCII ist es so, dass jedes Zeichen durch ein Oktett, also quasi eine Folge von acht Einsen bzw. Nullen, repräsentiert wird. Das ergibt rechnerisch eine Gesamtmenge von 128 möglichen Zeichen. Bei UTF-8 hingegen werden Zeichen auch durch Oktettkombinationen repräsentiert, was zur Folge hat, dass das Repertoire an möglichen Zeichen deutlich größer ist.

Für historische Korpora des Deutschen wird in aller Regel UTF-8 verwendet, einfach weil der ASCII-Zeichensatz nicht ausreicht, um die zahlreichen Sonderzeichen darzustellen, deren es umso mehr gibt, je weiter man in frühere Sprachstufen zurückgeht. Wenn Unicode-Text als ASCII interpretiert wird, dann ist das so lange unproblematisch, wie er nur Zeichen enthält, die auch in ASCII enthalten sind. Sobald aber Sonderzeichen auftreten, werden sie nicht richtig eingelesen, wie wir im Screenshot in Fig. 2 bereits gesehen haben.

In Excel und Calc müssen wir beim Einlesen der Daten einfach nur die richtige Kodierung wählen. In R kann die Kodierung unter Umständen zu größeren Problemen führen, weshalb man sich mit den Argumenten *encoding* und *fileEncoding*, die sich in den Funktionen zum Einlesen und Ausgeben vom Daten wie z.B. *read.table* oder *read.delim* und *write.table* finden, und mit den Unterschieden zwischen diesen beiden vertraut machen sollte. Im Tutorial zu R kommen wir noch einmal darauf zurück.

Auch die Umlaute in R-Skripts können sich als problematisch erweisen, wenn man z.B. ein unter Linux/Unix erstelltes R-Skript auf einem Windows-PC öffnet oder umgekehrt. Deshalb habe ich versucht, in den R-Skripts im Begleitmaterial völlig auf Umlaute und <ß> zu verzichten und sie durch <oe>, <ae>, <ue> und <ss> zu ersetzen.

Die Universalgrammatik der Suchanfrage: Reguläre Ausdrücke

In der Sprachwissenschaft gibt es die (umstrittene) Theorie, dass allen Sprachen eine angeborene „Universalgrammatik“ zugrundeliegt. Mit Korpusabfragesprachen ist es nun einerseits ganz ähnlich wie mit echten Sprachen: Ungeachtet aller Gemeinsamkeiten, die viele oder möglicherweise gar alle verbinden, gibt es eine beachtliche Heterogenität, weshalb man eine Korpusabfragesprache, genau wie eine natürliche Sprache oder eine Programmiersprache, zunächst einmal lernen muss. Andererseits – und das ist die gute Nachricht – hält sich die Heterogenität bei Korpusabfragesprachen trotz aller Unterschiede in halbwegs überschaubaren Grenzen. Und es gibt tatsächlich so etwas wie eine „Universalgrammatik“, von der nahezu alle Korpusabfragesysteme Gebrauch machen, nämlich **reguläre Ausdrücke**.

Reguläre Ausdrücke werden zum Auffinden bestimmter Zeichen bzw. Zeichenfolgen benutzt (vgl. Fitzgerald 2012). Leider ist es auch bei regulären Ausdrücken nicht so, dass es *ein* festes Inventar gibt, sondern es gibt auch hier subtile Unterschiede – z.B. gibt es verschiedene Standards, die sich aber z.T. ergänzen; für unsere Zwecke werden wir über diese Unterschiede komplett hinwegsehen, aber es ist wichtig zu wissen, dass eben deshalb nicht alle Korpusabfragesysteme alle regulären Ausdrücke unterstützen. Teilweise liegt das auch nicht an unterschiedlichen Standards, sondern hat vermutlich Kapazitätsgründe – wenn man z.B. im DWDS nach `$w=/.*/` sucht (ein absolut valider regulärer Ausdruck, der bedeutet: „Gib mir alle Tokens, in

denen auf ein beliebiges Zeichen 0 oder beliebig viele weitere Zeichen folgen“), bekommt man folgende Fehlermeldung:

Fehlermeldung
zu viele sinnfreie Regexes (max=0)

Das „Cheat Sheet“ auf den nächsten Seiten, das sich nach meiner Erfahrung auch gut für den Unterricht eignet, fasst die wichtigsten regulären Ausdrücke zusammen, darunter auch **lookaround assertions**, die dann hilfreich sind, wenn ich z.B. einen Tabstopp (</t>) oder Zeilenumbruch (</n>) vor oder nach einem bestimmten String einfügen möchte, ohne diesen String zu ersetzen. Einfaches Beispiel: Ich habe ein Textdokument mit Sätzen, in denen Komposita mit dem Bestimmungsglied *-landschaft* vorkommen, z.B. (aus DECOW, Schäfer & Bildhauer 2012):

Jürgen Habermas liefert in diesem Essay in der Süddeutschen vom 16.05.2007 eine wohl zutreffende Krisenanalyse zu den jüngsten Entwicklungen in der überregionalen **Zeitungslandschaft**.

denn in den Jahrzehnten seit seiner Fertigstellung wurde Pantons ursprüngliche **phantasielandschaft** im Eingangsbereich, in den Büros und in den Besprechungsräumen nach und nach durch immer konservativere Töne und Formen ersetzt.

Absolute Ruhe und Erholung: Das über 100 Jahre alte Anwesen "Zur Strandburg" liegt inmitten der traumhaften Rügener **Naturlandschaft** mit einmaligem Blick auf die tiefblaue Ostsee.

Um diese Konkordanz im KWIC-Format in ein Tabellenkalkulationsprogramm einlesen zu können (vgl. dazu das nächste Tutorial „02-Arbeitsschritte“), möchten wir nun vor und nach dem Kompositum auf *-landschaft* einen Tabstopp einsetzen. Das geht, indem wir zunächst *landschaft* durch *landschaft\t* ersetzen (das ist der einfache Teil) und anschließend

```
(?=[[:space:]]+([[:alnum:]]*landschaft)
```

durch *\t* ersetzen. Diese komplexe Lookahead Assertion findet die Stelle, auf die zunächst ein Leerzeichen und dann *-landschaft* folgt, mit beliebig vielen alphanumerischen Zeichen dazwischen. Das Ganze sieht dann so aus (hier zeige ich beispielhaft nur einen der Belege; die Tabstopps sind grün hervorgehoben):

denn in den Jahrzehnten seit seiner Fertigstellung wurde Pantons ursprüngliche **phantasielandschaft** im Eingangsbereich, in den Büros und in den Besprechungsräumen nach und nach durch immer konservativere Töne und Formen ersetzt.

Das ist zugegebenermaßen extrem komplex, und Sie müssen nicht erschrecken, wenn Sie das Beispiel nicht verstehen! Aber wenn Sie sich ein wenig damit auseinandersetzen, werden Sie feststellen, dass das Ganze wunderbar logisch ist, auch wenn es eine Weile dauert, bis man die Logik durchschaut hat.

Die wichtigsten regulären Ausdrücke

grau hinterlegt: Beispiele

Boolesches "oder"

| "oder" (Alt Gr + </> unter Windows, Alt+7 bei Mac)

Fass|Fass lies: <Fass> ODER <Faß> - sucht nach beiden Schreibvarianten

Gruppierung durch Klammern

() Runde Klammern definieren eine **Erfassungsgruppe** (*capturing group*)
[] Eckige Klammern definieren eine **Zeichenklasse** (*character class*), z.B. [abc] = irgendein Zeichen aus dem Inventar a,b,c, [asdf] irgendein Zeichen aus dem Inventar a,s,d,f.
[^] quasi das negative "Gegenstück" zu []: irgendein Zeichen, das **nicht** in dem Inventar an Zeichen enthalten ist, das in den eckigen Klammern definiert wird, z.B. [^abc]: irgendein Zeichen, das nicht a, b oder c ist.

Fa(ss|ß) sucht nach <Fass> ODER <Faß>
[HML]aus sucht nach <Haus>, <Maus>, <Laus>.
[^F]önig sucht nach allen Wörtern, bei denen der Zeichenfolge -önig ge-
nau ein Zeichen vorangeht, das jedoch nicht F sein darf. *König* wird
also gefunden, *Fönig* hingegen nicht (sofern Letzteres überhaupt
im Forpus belegt ist).

Wildcard (Platzhalter)

. irgendein Zeichen

.aus findet *Haus*, *Maus*, aber nicht *Graus*: . steht für genau **ein** Zeichen

Quantifikation

? das Zeichen unmittelbar davor tritt 0- oder 1-mal auf.
* das Zeichen unmittelbar davor tritt 0- oder x-mal (in unmittelbarer Folge) auf.
+ das Zeichen unmittelbar davor tritt 0- oder x-mal (in unmittelbarer Folge) auf.
{n} das Zeichen unmittelbar davor tritt genau n-mal (in unmittelbarer Folge) auf.
{x,} das Zeichen unmittelbar davor tritt mindestens x-mal (in unmittelbarer Folge) auf.
{x,y} das Zeichen unmittelbar davor tritt mindestens x-, maximal y- (in unmittelbarer Folge) auf.
mal

das? findet <das> und <dass>
da(ß|ss?) findet <das>, <dass> und <daß>
ne.*in findet <nein>, <neein>, <neeein>, <neeeein> etc. etc.
(M|m)an.? findet <man>, <Man> <Mann>, <mann>, <Mans>, <Manu> etc.

(M m)an.{1,5}	findet <man>, <Mann>, <manch>, <Mangel> etc., aber nicht <Mannschaft> (weil sieben Zeichen auf das <i>Man</i> folgen)
---------------	---

Anfang und Ende

^	Anfangsposition
\$	Endposition

^S	findet alle Wörter, die mit <S> anfangen.
es\$	findet alle Wörter, die auf <es> enden.

Escape Strings

\	Escape-Zeichen
---	----------------

"Hallo\?\!?\\"	findet <"Hallo?!?"> (in Anführungszeichen & mit "echten" !?!))
----------------	---

Klammerausdrücke

Mit einer Reihe von Klammerausdrücken lassen sich z.B. Interpunktionszeichen, numerische oder nicht-numerische Zeichen finden. Sie lassen sich mit einem ^ negieren, d.h.

[[:alnum:]]	findet alphanumerische Zeichen
[^[:alnum:]]	findet nicht-alphanumerische Zeichen

[[:alpha:]]	alphabetische Zeichen (keine Zahlen)
[[:digit:]]	Ziffern
[[:blank:]]	Leerzeichen oder Tabstopps
[[:punct:]]	Interpunktion und Symbole

Wo ist was auf der Tastatur?

Zeichen	Windows	Mac
	Alt Gr + </>	Alt + 7
[Alt Gr + 8	Alt + 5
]	Alt Gr + 9	Alt + 6
{	Alt Gr + 7	Alt + 8
}	Alt Gr + 0	Alt + 9
\	AltGr + ?/B	Alt+Shift+7
/	Shift+7	Shift+7

Lookaround Assertions

(?=foo)	Lookahead	Der String, der der gesuchten Position unmittelbar folgt, ist <i>foo</i>
(?<=foo)	Lookbehind	Der String, der der gesuchten Position unmittelbar vorausgeht, ist <i>foo</i>
(?!foo)	Negative Lookahead	Der String, der der gesuchten Position unmittelbar folgt, ist nicht <i>foo</i>
(?<!foo)	Negative Lookbehind	Der String, der der gesuchten Position unmittelbar vorausgeht, ist nicht <i>foo</i>

Lookaround Assertions haben den Vorteil, dass man Suchen & Ersetzen verwenden kann, ohne tatsächlich etwas zu ersetzen. Angenommen beispielsweise, wir haben eine Liste mit unterschiedlich langen Textabschnitten und wollen vor das letzte Wort in jeder Zeile einen Tabstopp einfügen. Das geht mit: Ersetze `(?=([^]*\n))` durch `\t`, wobei `\n` für „neue Zeile“ steht und `\t` für „Tabstopp“. Eine konkrete Anwendungsmöglichkeit hierfür ist z.B., wenn wir eine Konkordanz im KWIC-Format haben, in der linker Kontext, Keyword und rechter Kontext jeweils in einer Spalte dargestellt werden, und wissen wollen, was das jeweils letzte Wort in der ersten Spalte ist (z.B. um Komposita zu finden, die ja in manchen Sprachen, etwa dem Englischen, oft ein Leerzeichen zwischen den Kompositionsgliedern haben).

Wichtig für Lookbehind: Hier müssen die Strings eine feste Länge haben, d.h. eine Anfrage wie `(?<=.*landschaft)` ist nicht möglich, weil der reguläre Ausdruck ja nach 0 bis beliebig vielen Wiederholungen des Platzhalters `.` sucht; in diesem Fall bekommen Sie z.B. in Texteditoren eine Fehlermeldung wie „lookbehind assertion is not fixed length“. Möglich ist hingegen so etwas wie `(?<=.{6}landschaft)`, da hier konkret angegeben ist, wie oft die Wildcard `.` wiederholt wird (nämlich 6-mal).

Geiz ist geil: Wie man reguläre Ausdrücke „non-greedy“ macht

Die regulären Ausdrücke, die wir hier kennengelernt haben, sind sehr mächtig – manchmal zu mächtig. Angenommen etwa, wir haben folgenden HTML-Code:

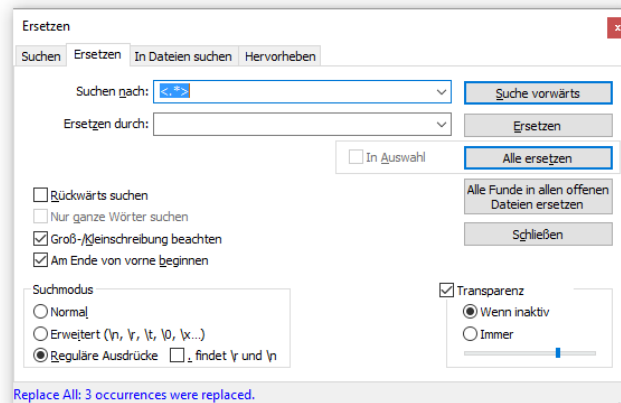
```
<a href="https://github.com/hartmast/">Stefans GitHub</a>
<a href="https://github.com/hartmast/sprachgeschichte">Sprachgeschichte</a>
<a href="https://github.com/hartmast/weltherrschaft">Geheimer Masterplan zur Weltherrschaft</a>
```

und wir wollen alles ersetzen, was in spitzen Klammern steht. In diesem Fall wäre es fatal, einen regulären Ausdruck wie `<.*>` zu verwenden, z.B. im „Ersetzen“-Dialog von Notepad++:

```

1 <a href="https://github.com/hartmast/">Stefans GitHub</a>
2 <a href="https://github.com/hartmast/sprachgeschichte">Sprachgeschichte</a>
3 <a href="https://github.com/hartmast/weltherrschaft">Geheimer Masterplan zur Weltherrschaft</a>
4

```



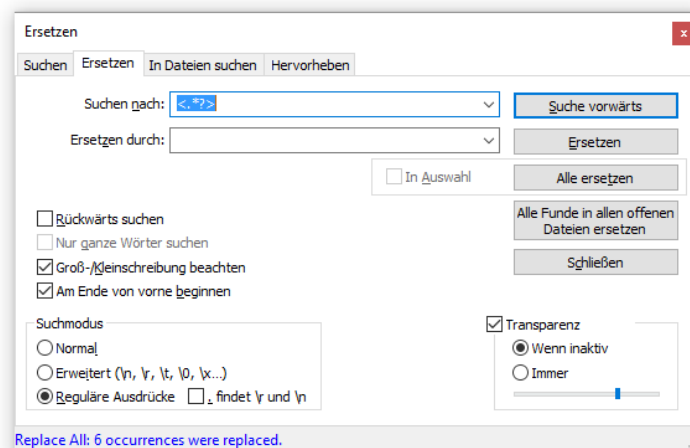
Übrig bliebe nämlich – nichts. Der reguläre Ausdruck ist **greedy**, er sucht alles bis zum letzten möglichen String, und in diesem Dokument ist der letzte mögliche String auch der letzte String im Dokument überhaupt. Was wir wollen, ist, dass der reguläre Ausdruck eine sog. *lazy evaluation* vornimmt. Das erreichen wir, indem wir dem Platzhalter `*` ein `?` folgen lassen (das wir oben schon in anderer Funktion kennengelernt haben; so ein Fragezeichen ist eben multitaskingfähig...)

Mit der „lazy“ Variante `<.*?>` bleibt genau das übrig, was übrig bleiben soll:

```

Stefans GitHub
Sprachgeschichte
Geheimer Masterplan zur Weltherrschaft

```



Übungsaufgaben

Bitte formulieren Sie folgende Suchanfragen mit Hilfe von regulären Ausdrücken (Sie können sie in einem Editor wie Notepad++ ausprobieren, indem Sie selbst einen kurzen Text schreiben, in dem die gesuchten Wörter vorkommen):

- *Fett* als Substantiv (groß geschrieben) und *fett* als Adjektiv (klein geschrieben).
- <König> oder <Königin>
- <Haßprediger> und <Hassprediger> inklusive der movierten Formen <Haßpredigerin> und <Hasspredigerin>
- Komposita mit dem Erstglied *Welt*-, z.B. *Weltkrieg*.
- <Student>, <Studentin>, <Studenten>, <Studentinnen>, <Studierende>, <Studierender>.
- Bitte schreiben Sie in einen Texteditor, der reguläre Ausdrücke unterstützt (z.B. Windows: Notepad++; Mac: TextWrangler): *Dies ist ein Test, der testen soll, wie Lookahead und Lookbehind funktionieren*. Benutzen Sie die Suchfunktion und Lookahed, um das Wort *schöner* vor *Test* einzufügen. Benutzen Sie Lookahead, um einen Tabstopp (\t) vor jedem Satzzeichen ([[:punct:]]) einzufügen.

Literatur

Fitzgerald, Michael. 2012. *Introducing regular expressions*. Sebastopol: O'Reilly.

Schäfer, Roland & Felix Bildhauer. 2012. Building Large Corpora from the Web Using a New Efficient Tool Chain. In Cicoletta Calzolari, Khalid Choukri, Terry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk & Stelios Piperidis (eds.), *Proceedings of LREC 2012*, 486–493.