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

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.

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 und 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 Arbeitsblatt „Die wichtigsten regulären Ausdrücke“), die man zwar de facto im Texteditor nur selten braucht, die aber trotzdem sehr nützlich sein können.

hier einfügen: Vergleich win-Texteditor mit notepad

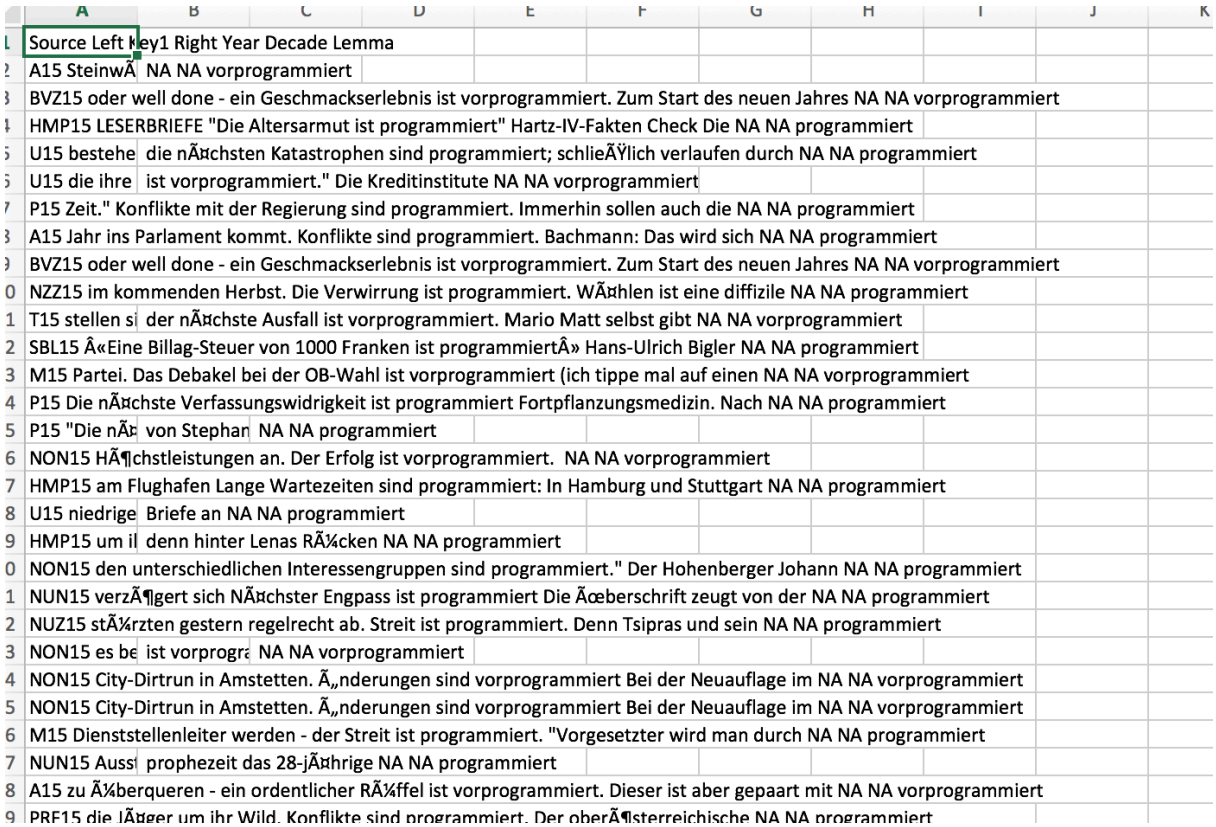
2. Tabellenkalkulationsprogramme: Excel und Calc

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 wenigers vns allen zu Trost/ **sterckung** deß Glaubens/ vnd zur besserung vnsers Sündhafftigen lebens gereichen möge/	NOBD-1620-KT-081	O	(zu)				
2.	wir auch nichts wenigers vns allen zu Trost/ sterckung deß Glaubens/ vnd zur **besserung** vnsers Sündhafftigen lebens gereichen 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	(X)	(neg				

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 SteinwÃ	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 nÃ¤chsten Katastrophen sind programmiert; schlieÃŸlich 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. WÃ¤hlen ist eine diffizile NA NA programmiert										
11	T15 stellen s	der nÃ¤chste Ausfall ist vorprogrammiert. Mario Matt selbst gibt NA NA vorprogrammiert									
12	SBL15 Â«Eine Billag-Steuer von 1000 Franken ist programmiertÂ» 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 nÃ¤chste Verfassungswidrigkeit ist programmiert Fortpflanzungsmedizin. Nach NA NA programmiert										
15	P15 "Die nÃ¤	von Stephan	NA NA programmiert								
16	NON15 HÃ¶chstleistungen 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 RÃ¼cken NA NA programmiert									
20	NON15 den unterschiedlichen Interessengruppen sind programmiert." Der Hohenberger Johann NA NA programmiert										
21	NUN15 verzÃ¶gert sich NÃ¤chster Engpass ist programmiert Die Ã¤berschrift zeugt von der NA NA programmiert										
22	NUZ15 stÃ¼rzten gestern regelrecht ab. Streit ist programmiert. Denn Tsipras und sein NA NA programmiert										
23	NON15 es be	ist vorprogra	NA NA vorprogrammiert								
24	NON15 City-Dirtrun in Amstetten. Ã„nderungen sind vorprogrammiert Bei der Neuauflage im NA NA vorprogrammiert										
25	NON15 City-Dirtrun in Amstetten. Ã„nderungen 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 Auss	prophezeit das 28-jÃ¤hrige NA NA programmiert									
28	A15 zu Ã¼berqueren - ein ordentlicher RÃ¼ffel ist vorprogrammiert. Dieser ist aber gepaart mit NA NA vorprogrammiert										
29	PRF15 die JÃ¤ger um ihr Wild. Konflikte sind programmiert. Der oberÃ¶sterreichische 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.

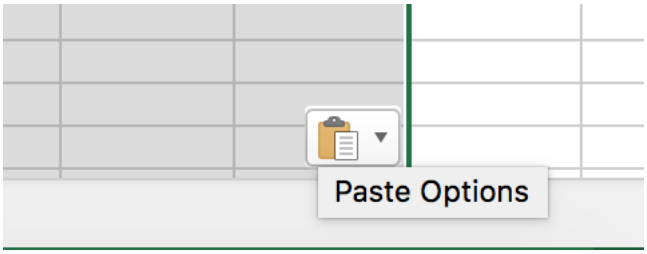


Fig. 3: Der kleine Button mit Einfügeoptionen (siehe Schritt 6)

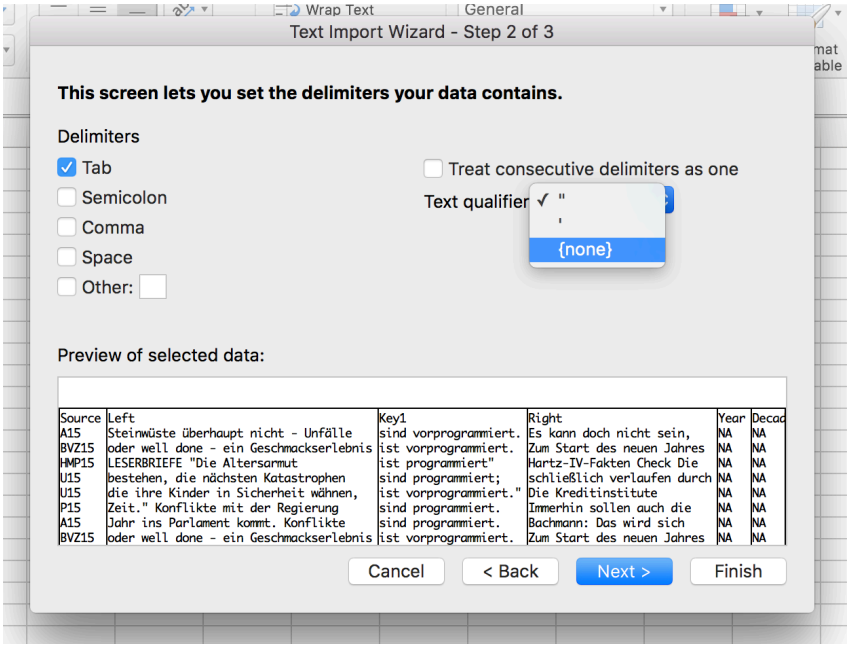


Fig. 4: Menü zur Auswahl von Texttrennern und Textqualifizierern.

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 „Öffen 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.

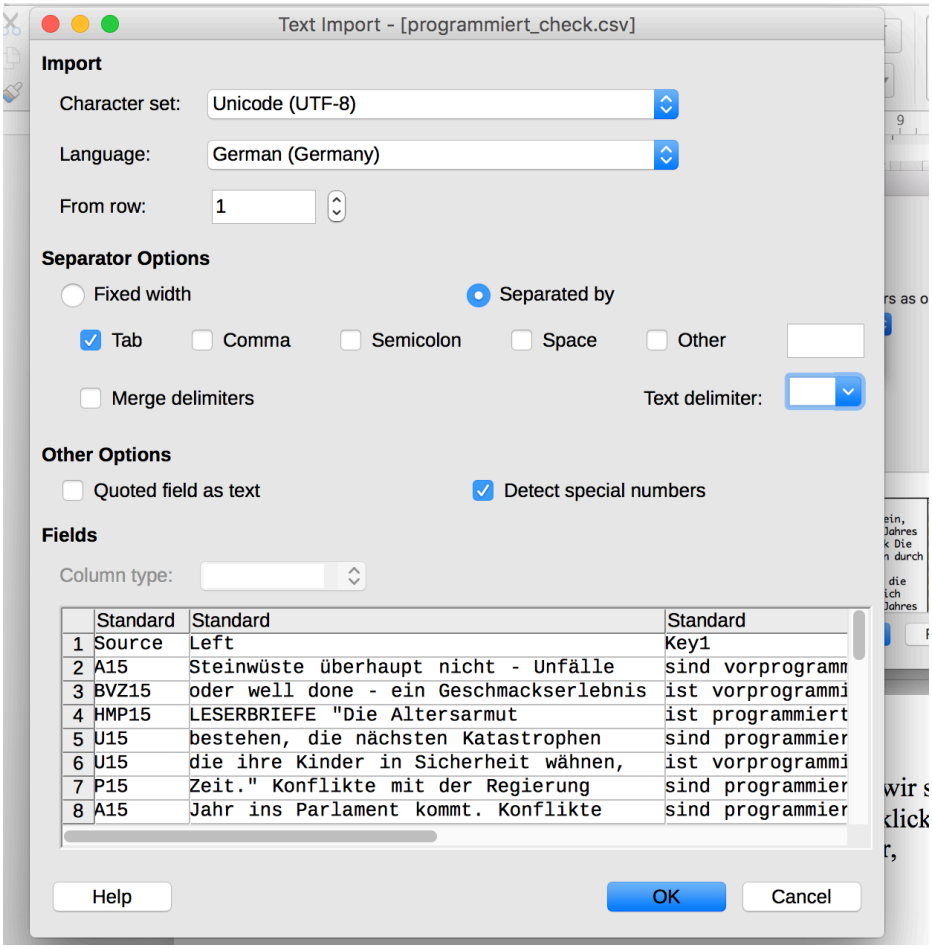


Fig. 5: Textimport-Assistent von LibreOffice Calc.

Praktische Übung
Lesen Sie die Datei *programmiert.csv*, die sich im Begleitmaterial findet, 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.)

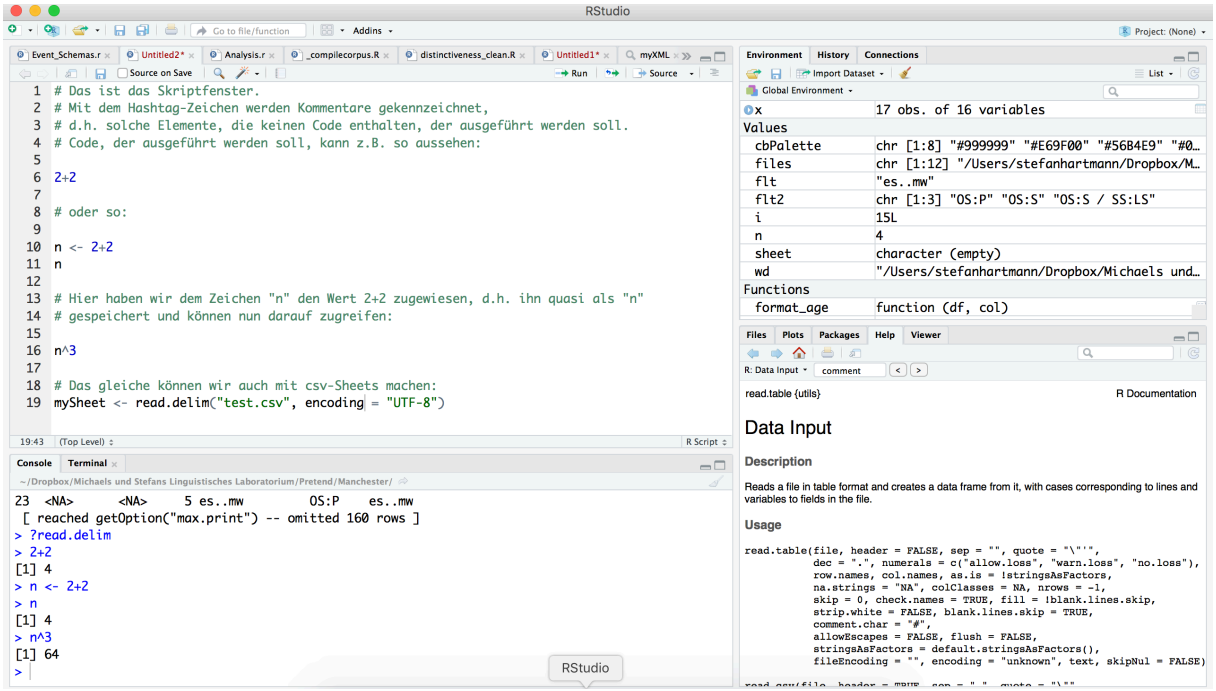


Fig. 6: Interface von RStudio. Oben links: Das Skriptfenster, unten links; Die Konsole; oben rechts: Das Environment; unten rechts: Die Hilfe, die man aufrufen kann, indem man aufrufen kann, indem man einen Funktionsnamen eingibt, dem ein Fragezeichen vorangestellt ist, z.B. ?read.delim.

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-Betriebsssytemen 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 versehentlich 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 *stringsAsFacors = 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, 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

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. 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 daran, dass in der „deutschen“ Notation das Komma als Dezimaltrennzeichen fungiert (z.B. 0,005 - null komma null null fünf); im englischen Sprachraum (und übrigens auch in R!) erfüllt diese Funktion der Punkt.

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 weiter unten: „Das geheime Leben meines Computers“) und haben Excel ein Schnippchen geschlagen. Bei Calc müssen wir diesen Umweg nicht gehen, sondern können das Trennzeichen beim Speichern direkt auswählen.

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 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. In den praktischen Beispielen kommen wir noch einmal darauf zurück.