DGS-KORPUS

Data Visualisation in iLex

Edited by: Thomas Hanke, 2013-12-01
Thomas Hanke, 2014-06-24: Some typos fixed
Thomas Hanke, 2016-03-30: Some typos fixed

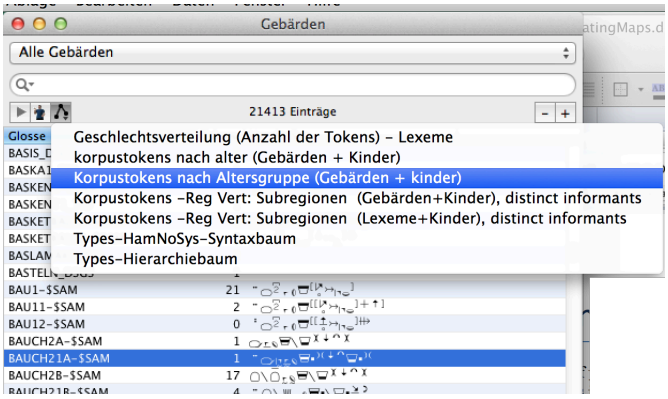
Introduction

If you want to create fancy visualisations of your final data in iLex, one easy way is to run your query and copy & paste the results into e.g. an Excel sheet, allowing you to use any chart styles that Excel provides. However, if you need to recreate these graphics regularly as the data change, you are far better off with using iLex-internal charting capabilities. Currently, iLex provides three kinds of graphics:

- Business charts such as pie charts (rendered iLex-internally using the *ChartDirector* library)
- Graphs (nodes and edges between them) (rendered on the client computer or on the server side using *graphviz*)
- Maps (rendered on the client computer or on the server side using *R*)

You can produce other types of charts by setting up appropriate *R* functions and defining corresponding `chart_styles` data.

All kinds of charts are either global or operate on one (or more) specific data set. All global charts are available via the “Charts” submenu of the “Data” menu. Charts requiring an input are available from the Charts menu button in the corresponding list window:

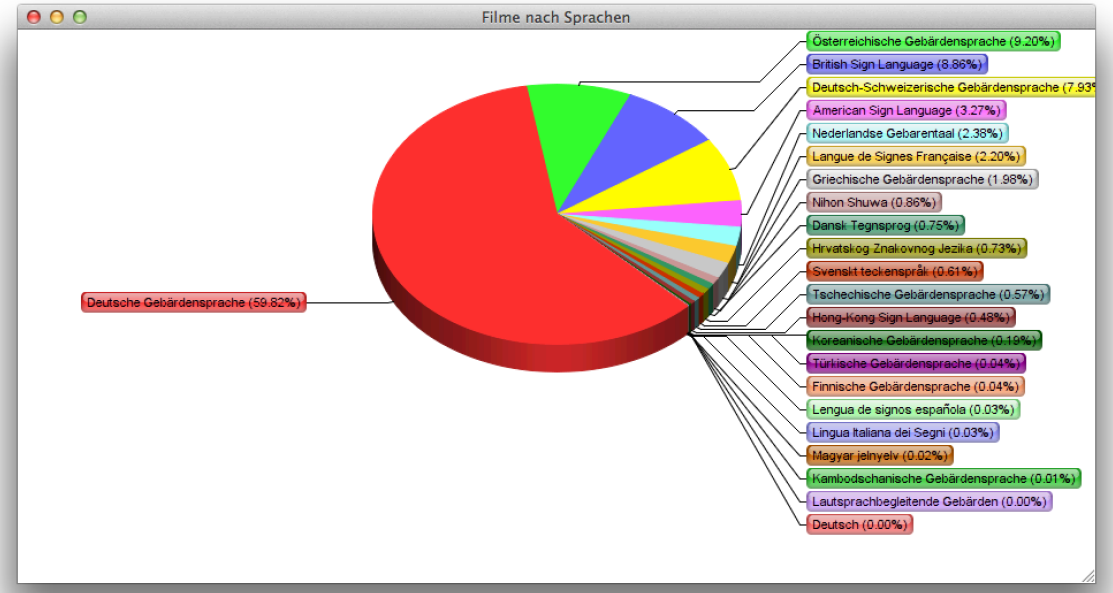


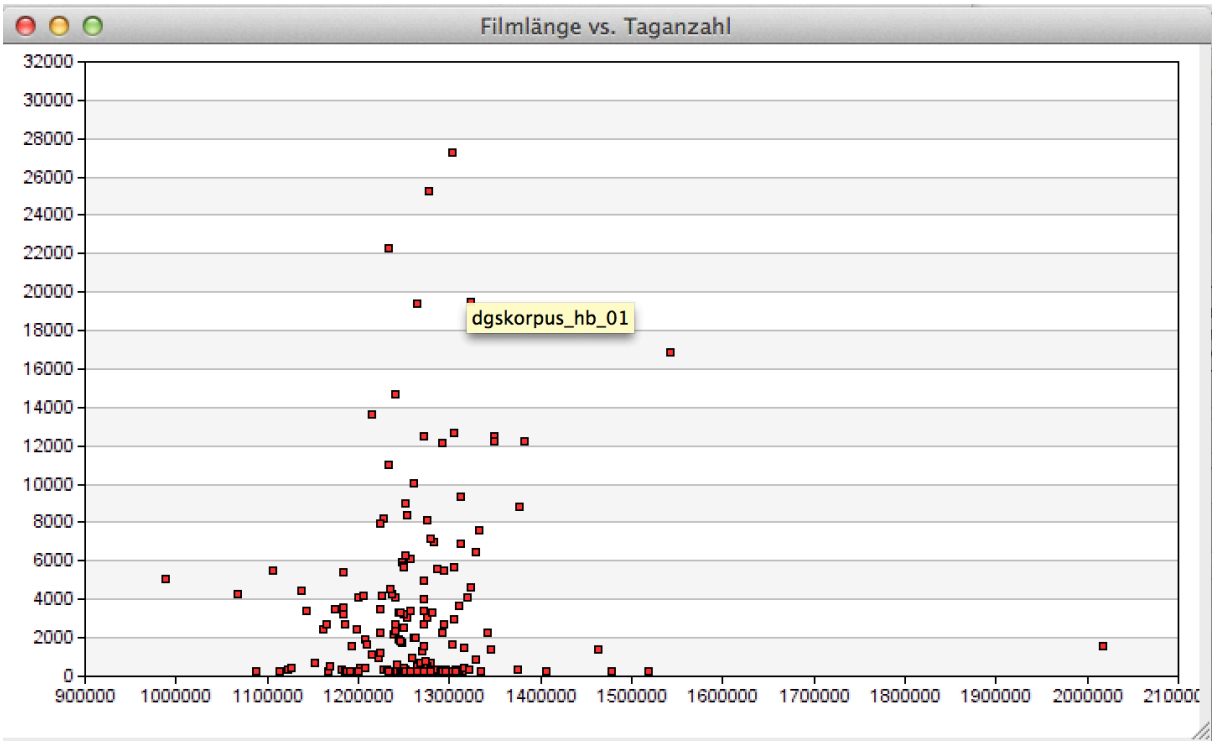
All chart definitions can provide a number of seconds how often the chart shall be updated while the chart window is open. Keep in mind that charting requires an SQL query and local processing so do not ask for very short update intervals.

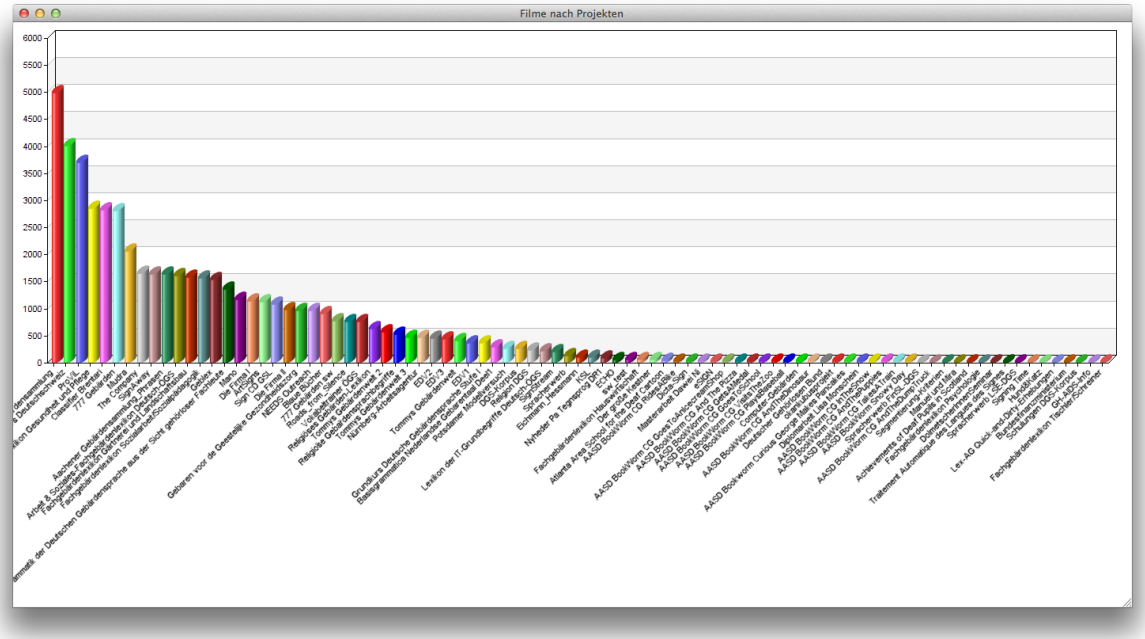
Charts can be saved as high-quality SVG images to be used presentations, publications etc.

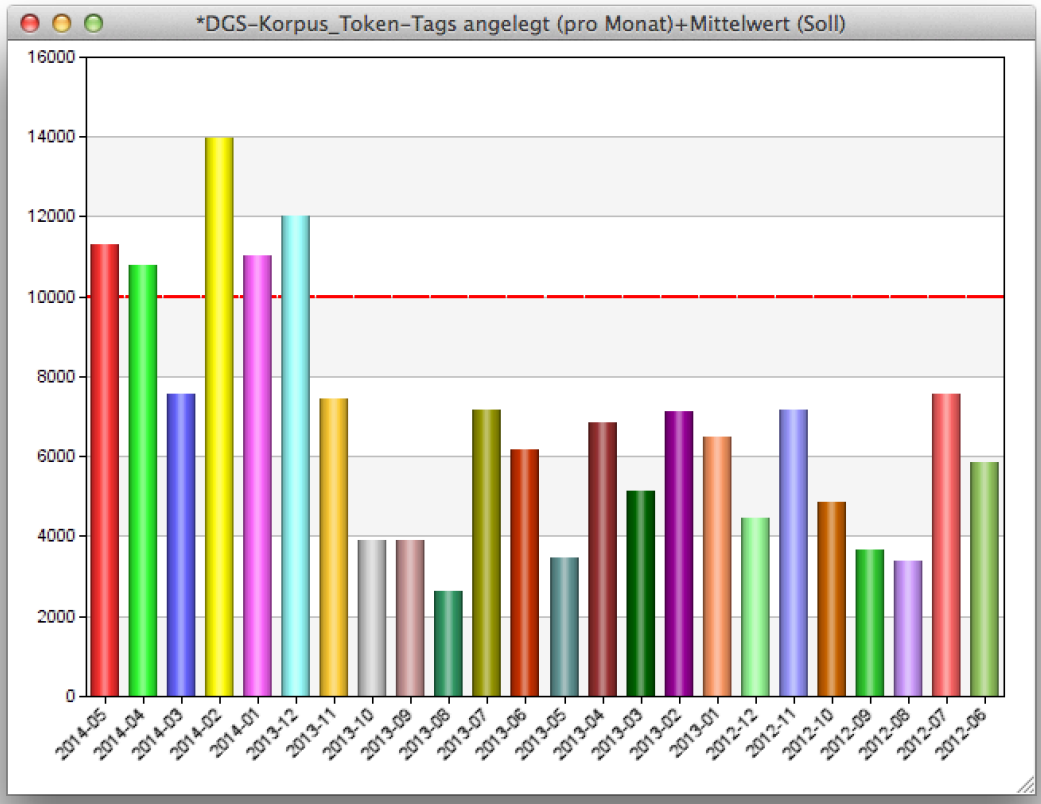
Business charts

A number of predefined business charts are available, pie charts, scatter plots, bar charts and bar charts with limits.

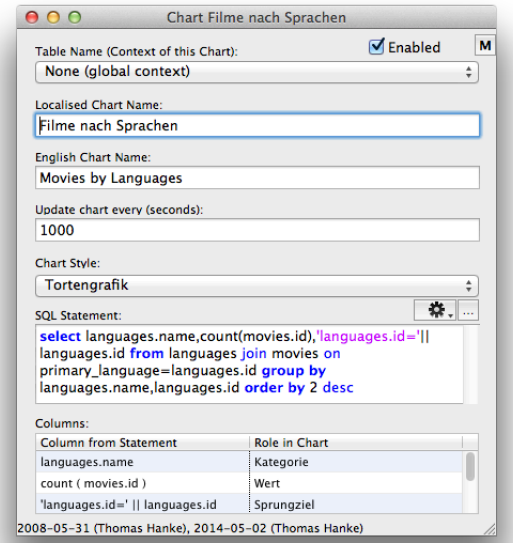

Filme nach Sprachen
Österreichische Gebärdensprache (9.20%)
British Sign Language (8.86%)
Deutsch-Schweizerische Gebärdensprache (7.93%)
American Sign Language (3.27%)
Nederlandse Gebarentaal (2.38%)
Langue de Signes Française (2.20%)
Griechische Gebärdensprache (1.98%)
Nihon Shuwa (0.86%)
Dansk Tegnsprog (0.75%)
Hrvatskog Znakovnog Jezika (0.73%)
Svenskt teckenspråk (0.61%)
Tschechische Gebärdensprache (0.57%)
Hong Kong Sign Language (0.48%)
Koreanische Gebärdensprache (0.19%)
Türkische Gebärdensprache (0.04%)
Finnische Gebärdensprache (0.04%)
Lengua de signos española (0.03%)
Lingua Italiana dei Segni (0.03%)
Magyar jelnyelv (0.02%)
Kambodschanische Gebärdensprache (0.01%)
Lautsprachbegleitende Gebärden (0.00%)
Deutsch (0.00%)
Deutsche Gebärdensprache (59.82%)


Filmlänge vs. Taganzahl
dgskorpus_hb_01
32000
30000
28000
26000
24000
22000
20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
900000
1000000
1100000
1200000
1300000
1400000
1500000
1600000
1700000
1800000
1900000
2000000





The structure of the datasets to be returned by the SQL query depends on the chart type. E.g. for pie charts, three columns are expected:

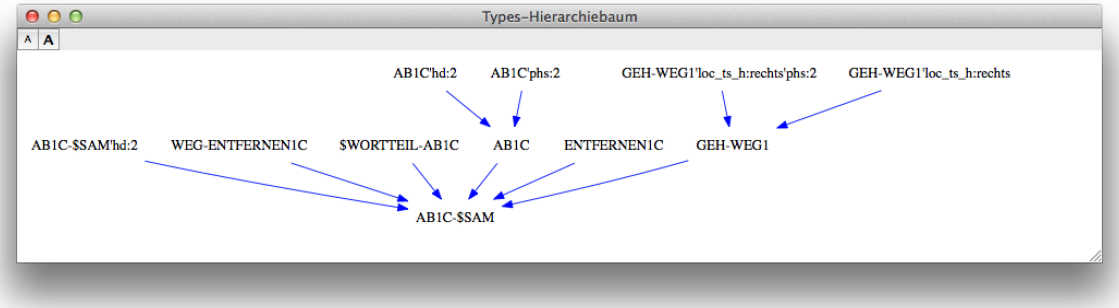


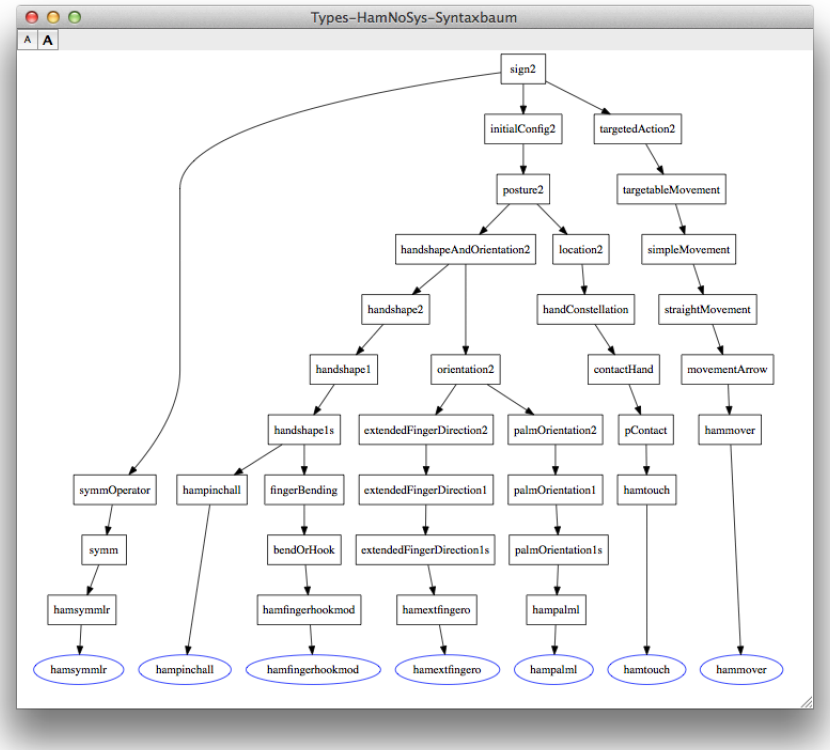
- The name of the data set to be show in the key labels of the pie chart
- The value for that data set (Normalisation to 100% for the sum of all values is done automatically)
- Optionally, a URL to be followed should the user click on the pie segment for this data set (This can be an http link, but also any iLex-internal link such as `ilex://languages.id=XXX`.)

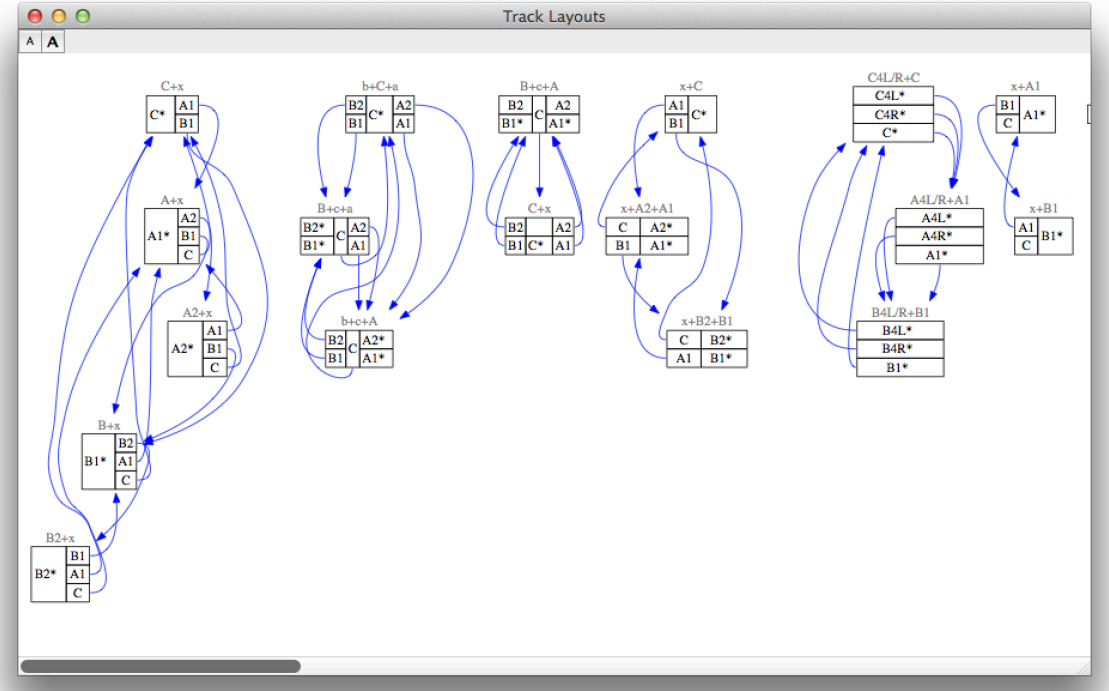
In graphs where the label is not visible by default you can hover over a specific data point to make the label visible (cf. the scatter diagram above).

The library used to render these charts (*ChartDirector*) is quite versatile. So if you need a business chart style not currently available in iLex, please file a feature request.

Graphs





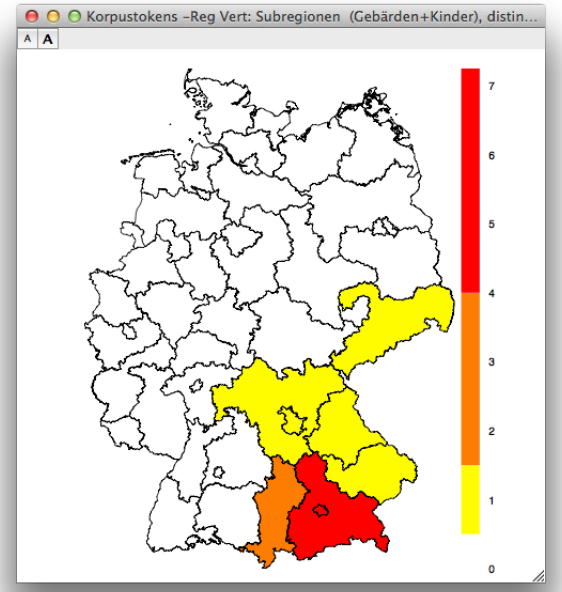


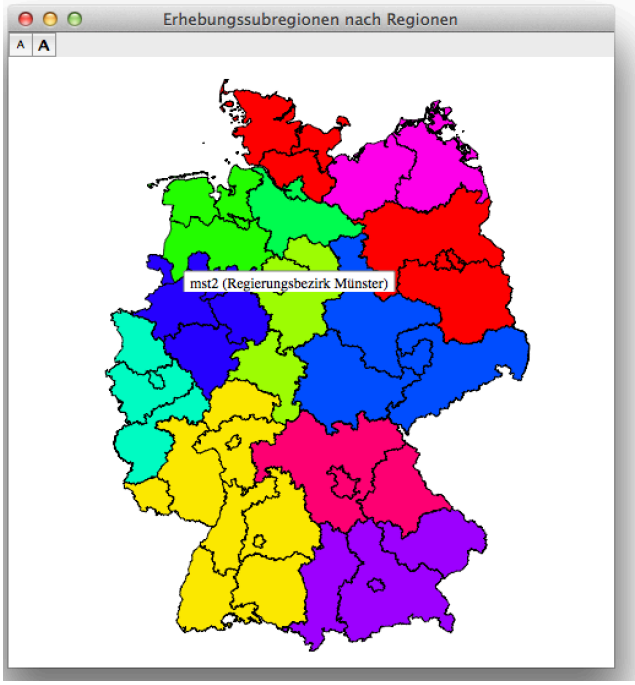
iLex uses *graphviz* to render graphs, i.e. sets of nodes with edges between them. All *graphviz* layout algorithms are available as chart styles in iLex. The chart_style's SQL statement needs to return a *graphviz* dot file. As such a file is easier to produce by a procedural language, it is recommended that you define an SQL function in a language such as Perl to process the input and create the dot output.

Graphs can show relations between different database entries in iLex or show the structure of complex entries. The dot format allows nodes and edges to have tooltips or URLs associated.

Creating map charts in iLex

With iLex, you can create maps based on any data somehow linked to geographical data. E.g. you may want to plot the regional distribution of a specific variant of a sign.





Internally, iLex uses *R* to render these maps, either remotely on the server or on your own desktop computer, depending on where the *R* system is installed.
The first step is to choose a map divided into regions. In the DGS Korpus project, we use four different maps of Germany:

- Germany divided into federal states

- Germany divided into counties (This is a further subdivision of the federal states.)
- The data collection regions
- The subregions for the data collection (a region consists of 2-5 subregions that are more or less a bunch of counties)

(We used official map data for Germany to produce these customised formats – see the appendix for a detailed description of the approach taken.)

The second step is to write an SQL query that will fill the map. The query should return four columns:

- ID of the region to be shown
- A value for this region
- A label for this region that is shown when the user moves the mouse over this region.
- A hypertext reference that will be opened if the user clicks on that region. (This can be an http link, but also any iLex-internal link, so you can e.g. open the variant of a sign that is the most frequent in the region clicked on.

ID

Only those regions will be drawn in the map whose IDs are contained in the query results. So if you only want to show the counties within one state, just restrict the IDs to those counties belonging to the state: The map will scale to fill the full drawing size, i.e. you then have a map of that state divided into counties.[1]

Values

You can provide any range of numerical data associated with the regions. The map will be coloured from yellow (minimum value) to red (maximum value) and have a key to identify values for specific colours. However, if 0 is the minimum value in the query result, all regions having this value will be shown in white.
You can also provide colours to draw each region. Just make the query return strings in the form #rrggbb (hex encoding). In this case, no key is included.
If you provide non-numerical values, the map will be coloured with as many colours as there are distinct values in the data. Or the other way round, regions sharing the same value will have the same fill colour. As a shortcut to use distinct colours for all regions plotted, use TRUE as the value for all value data.

Labels and Targets

The latter two columns can be filled with NULL if you do not need them (i.e. they do not make sense for your map.) For iLex-internal hyperlinks you can leave out the `ilex://` prefix, i.e. a link `languages.id=1` is treated as `ilex://languages.id=1`. As it is possible to render maps from a URL by using a URL like `ilex://charts.id=1&action=render&param=2`, clicking on one region of a map may also open a detailed map of that region.

Handling URLs

On MacOS, iLex registers a scheme handler for `ilex://`, i.e. all URLs starting with `ilex://` in a browser or wherever are handed to iLex. iLex handles two kinds of URLs:

[1] In the DGS Korpus project, the IDs can be found in the vocabularies georegions/states, georegions/counties, georegions/regions and georegions/subregions.

- `ilex://select%20name%20from%20types%20where%20name~'^AB'` and such: Any URL-encoded SELECT statement is executed and the results are displayed in a Free Query window. Please note that for security reasons only SELECT statements are allowed.
- `ilex://types.name='AB1A'` and such: iLex opens a detail window for any of its main tables to display the specified data set.
  For some tables, adding a second parameter in the form `ilex://charts.id=1&action=render` does not open the detail window, but displays the execution of the specified data set. This shows a video snippet for types, tags, tokens, movies, it renders the chart for charts, or opens a list window for filters. For some charts as well as elements of the lists table, the URL needs to have a third parameter to provide the context, e.g. `ilex://charts.id=2&action=render&param=7` renders the chart with id 2 which takes an ID as context parameter, such as a types.id e.g. to render a graph of the that data set. For the table chart_styles, the extra parameter is an SQL `SELECT` statement the result of which is displayed using the defined chart style. In some cases, the extra parameter is supposed to be a list of IDs. In that case, just use a comma-separated list.

Notes:

- I am not aware that a dynamic registration of scheme handlers is possible in Windows, instead you need to add a registry key, cf. http://msdn.microsoft.com/en-us/library/aa767914%28v=VS.85%29.aspx
- Custom URL schemes currently do not work in Chrome.

Appendix: An Adventure on Maps for DGS-Korpus

Now that iLex can use R to produce neat map graphics, the question remains how to obtain appropriate map data for the DGS-Korpus data collection regions.
At this point of time, we are not interested in topographic data, but just into administrative borders also reflecting school district boundaries etc. The most prominent division to be reflected is the former West-Eastern boundary.
In the DGS-Korpus project, we have divided Germany into 13 regions, with up to five subregions for each region. As you can clearly see from the number, this division is not simply a reflection of the German states. Instead, some regions span state boundaries whereas others are subdivisions of one single state.
It first seemed that the data from gadm.org comes very close to what I had in mind. Most conveniently, they provide their data on different detail levels right in R dataframe format. However, when matching our data collection subregions with their county (Kreise and Kreisfreie Städte) data, I found that some changes to administrative units in East Germany as old as from 1998 were not reflected in the current dataset. Some of these were mentioned as bugs in their forum (but not marked to be fixed in the upcoming release 2.1) while others were not. Fusion is easy to handle, splitting is not. Moreover, these problems undermined my trust into this dataset, so I started looking into alternatives.
Fortunately, the Federal Government decided to make part of their geodata sets publicly available, and the first data are actually online since March 2013. This includes administrative borders, of course. The assumption is that this data is very accurate. The one in a more than sufficient resolution is called VG1000. As it uses official ids for counties etc., future data like the population census in 2020 is hopefully easy to relate to our data collection regions. As this source is open data now, the only requirement is mentioning the data source, which we would do anyway.
So, hands on. What needs to be done?

1. The data is provided in Shapefile(s) format. While not as convenient as an R dataframe format, reading the data in is straightforward.
2. The data contains two extra regions, the German parts of the North Sea and the Baltic Sea. As we did not include Deaf people living somewhere on a sand in the sea (islands of course belonging to some counties), there is no need for us to have these areas plotted. That is they can be eliminated from the plot.
3. Counties need to be aggregated to match our subregions.
4. There is one problematic case: Berlin is one single community with no subdivisions in the data provided whereas we have subdivided Berlin into two subregions (former West and East Berlin). So we need to split that region into two, introducing new polygon points.

I thought it would be the nicest to start with the county level so the other levels would be "simple" aggregation, so points 4 and 2 would need to be dealt with first.

Reinstantiating the Berlin Wall

Where do we get the polygon points for the West-East Division from? Googling resulted in a hit at the Berlin Open Data project that sounded promising: Just aggregating district polygons into the two parts should give the wanted result. The data is only available as kml, but by now I trusted there would be some nice R library to convert. While clicking the download link resulted in a 404 server error, googling the file name resulted in a hit at an official site, so I downloaded from there. kml is XML, so to have a first look at the data I opened it with Xmplify: 98 validation errors. Not that good. Maybe these errors are not important for my intended purpose, so: OGR is the name of an R library to import kml, let's try. Unfortunately, that package did not load, giving some obscure error messages that seemed to require quite some time to understand the background.

So the second best idea was to compile the points for the Berlin wall by manually examining the kml data. When two districts share a border, they should have the same sequence of coordinates (although one in reverse order). So I looked for pairs of districts that used to form the border, such as Mitte and Kreuzberg, Friedrichshain and Kreuzberg, Alt-Treptow and Kreuzberg etc. This resulted in the description of the wall by 1068 coordinate pairs. Far too much detail. How do we reduce that?

The file (named BerlinerMauer2.txt) looks like
13.36698401724482,52.62536834604824
13.36703838219619,52.62538212562937
13.36711099047662,52.62540052858895

(around 13.367° East, 52.625° North)

Get it a decent header and make it a closed polygon by adding
"Lon","Lat"
13.0,52.3
13.36698401724482,52.7
to the top of the file and
13.51600315221059,52.3
13.0,52.3
13.0,52.3
to the end. This gives a rectangle (except for the rough right side being the wall) which overlaps West Berlin. Then read the file and wrap the data into a SpatialPolygons object:

```
library(sp)
library(maptools)
library(rgdal)
library(rgeos)
berlin.wall.coord<-read.csv("BerlinerMauer2.txt")
berlin.wall.spdf<-SpatialPolygons(list(Polygons(list(Polygon(berlin.wall.coord)),1)))
```

Now we can use the Douglas-Peuker algorithm implemented by the thinnedSpatialPoly function, which indeed reduces the number of needed coordinates from 1068 to 187:

```
berlin.wall.thinned<-thinnedSpatialPoly(berlin.wall.spdf,0.0001)
```

Now, how do we combine that with the VG1000 data for Berlin? The surprise is that the VG1000 shape files do not provide the coordinates in degrees East and North, but instead in UTM, and to make things puzzling for the newcomer, in Zone 32 for all of Germany. That is, all converter services you find in the web provide completely different eastings for Berlin as Berlin is located in zone 33. Once again, there are helpful libraries, i.e. they are helpful if you know how to set up the parameters (took me an hour googling for a clue), so we convert our Berlin Wall plus rectangle to UTM zone 32 data:

```
west.wall<- berlin.wall.thinned[1]@polygons[[1]]@Polygons[[1]]@coords
west.wall.utm<-project (west.wall, "+proj=utm +zone=32 +ellps=GRS80 +units=m
+no_defs")
```

Make it a spatial data type again:

west.wall.spdf<-SpatialPolygons(list(Polygons(list(Polygon(west.wall.utm)),1)))
proj4string(west.wall.spdf)="+proj=utm +zone=32 +ellps=GRS80 +units=m +no_defs"

If we plot this, it looks as expected:

Before we can continue to extract West and East Berlin, we need to load the VG1000 data set:

counties0<-readShapeSpatial
("./vg1000_3112.utm32s.shape.kompakt/vg1000_kompakt/vg1000_f.shp",
IDvar="DEBKG_ID",proj4string=CRS("+proj=utm +zone=32 +ellps=GRS80 +units=m
+no_defs"))

Now we build the intersection (with 11000 being the county code for Berlin:

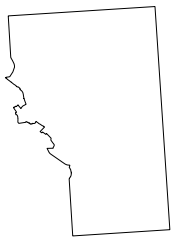
west.berlin<-gIntersection(counties0[which(counties0@data$RS==11000),],west.wall.spdf)

Now that looks as we wanted it to be:

Ok, now the same for the Eastern part: Set the rectangle straight side from 13.0° to 13.8°:

east.wall<- west.wall
east.wall[which(east.wall [,1]==13.0),1]=13.8
east.wall.utm<-project (east.wall, "+proj=utm +zone=32 +ellps=GRS80 +units=m
+no_defs")
east.wall.spdf<-SpatialPolygons(list(Polygons(list(Polygon(east.wall.utm)),1)))
proj4string(east.wall.spdf)="+proj=utm +zone=32 +ellps=GRS80 +units=m +no_defs"
east.berlin<-gIntersection(counties0[which(counties0@data$RS=="11000"),],east.wall.spdf)

This gives us



and

Looks perfect. If only it hadn't taken that much time…

While we are on it: There is another, although not that difficult related problem: For historical reasons, two islands in the North Sea belong to the State of Hamburg (county code 02000), not to Lower Saxony as one would suspect on the basis of their geolocations. While it does not really matter for the DGS-Korpus project (one island is uninhabited, and the other one probably has no deaf inhabitant, at least none participated in the data collection), it makes map drawing a bit odd as the Hamburg region appears much larger than it actually is due to the two outlier polygons. So for the purpose of this mapping, we just assign these islands to Lower Saxony's county Cuxhaven (county code 03352):

```
hamburg<-counties0[which(counties0@data$RS=="02000" & counties0@data$USE==4 &
counties0@data$GF==4),]
cuxhaven<-counties0[which(counties0@data$RS=="03352" & counties0@data$USE==4 &
counties0@data$GF==4),]
cux.mod<-SpatialPolygons(c(Polygons(c(cuxhaven@polygons[[1]]@Polygons[1:2],
hamburg@polygons[[1]]@Polygons[3:4]),"03352")),proj4string=cuxhaven@proj4string)
ham.mod<- SpatialPolygons(c(Polygons(c(hamburg@polygons[[1]]@Polygons[1:2]),
"02000")),proj4string=hamburg@proj4string)
```

(The additional conditions filter out water areas belonging to those counties.)
The same story applies to Heligoland, for some reasons part of the county of Pinneberg. We move it to Dithmarschen with is the closest county at the coast:

```
pinneberg<-counties0[which(counties0@data$RS=="01056" & counties0@data$USE==4
& counties0@data$GF==4),]
dithmarschen<-counties0[which(counties0@data$RS=="01051" &
counties0@data$USE==4 & counties0@data$GF==4),]
pin.mod<- SpatialPolygons(c(Polygons(c(pinneberg@polygons[[1]]@Polygons[1:2]),
"01056")),proj4string=pinneberg@proj4string)
```

```
dit.mod<-SpatialPolygons(c(Polygons(c(dithmarschen@polygons[[1]]@Polygons[],
pinneberg@polygons[[1]]@Polygons[3]),"01051")),
proj4string=dithmarschen@proj4string)
```

Please note that the state of Bremen also consists of two separate polygons (Bremen and Bremerhaven). As they are not that far from each other and belong to the same subregion in the data collection, no action needs to be taken here.

Building a custom county data frame

Basically, we now need to join all counties except Berlin, Hamburg and Cuxhaven, and join that with the data for West Berlin, East Berlin, and the manipulated data for Hamburg and Cuxhaven:

```
x<-spRbind(spChFIDs(west.berlin,c("11100")),spChFIDs(east.berlin,c("11200")))
x<-spRbind(ham.mod,x)
x<-spRbind(cux.mod,x)
x<-spRbind(pin.mod,x)
x<-spRbind(dit.mod,x)

counties1 <- counties0[which((counties0@data$USE==4) & (counties0@data$GF==4) &
(counties0@data$RS!= "11000") & (counties0@data$RS!= "02000") &
(counties0@data$RS!= "03352") & (counties0@data$RS!= "01051") &
(counties0@data$RS!= "01056")),]

for (i in 1:length(counties1)) { counties1@polygons[[i]]@ID =
as.character(counties1$RS[i])}

counties2<-as(counties1,"SpatialPolygons")
counties3<-spRbind(counties2,x)
```

counties3 is now the final SpatialPolygons list with all “counties” that we need. Now for the descriptions:

```
df<-read.table("Kreise_dataframe.txt",sep="\t",colClasses =
c("NULL","NULL","character","NULL","NULL","NULL","NULL","factor","factor","factor",
"character","character"),col.names=c("NULL","NULL",'Name',"NULL","NULL","NULL","
NULL",'Qualifier','Region','Subregion','ID','English'),header=FALSE,fill=TRUE)
for (i in df$ID) {n<-df[which(df$ID==i),"English"]; if (n=="")
{df[which(df$ID==i),"English"]=df[which(df$ID==i),"Name"]}}
row.names(df)=df$ID

counties <- SpatialPolygonsDataFrame(counties3,df)
```

As the last step, we save the data frame to an R file and we are done with one level of aggregation:

```
save(counties,file="counties.RData")
```

For testing, here is the easiest case:

```
col=rainbow(length(levels( as.factor(counties $Region))))
```

```
spplot(counties,"Region",col.regions=col,main="Erhebungsregionen")
```

Now to group counties into subregions and subregions into regions, we start with reading in the names of the regions and subregions:

```
names<-read.table("RegionsAndSubregions.txt", header=FALSE,
col.names=c("ID","Name","English"), sep="\t", colClasses =
c("character","character","character"), fill=TRUE)
row.names(names)<-names$ID
for (i in names$ID) {
        n<-names[which(names$ID==i),"English"]
        if (n=='') {
                names[which(names$ID==i),"English"]=
                        names[which(names$ID==i),"Name"]
        }
}
```

Now we loop through all subregions and for each union all counties belonging to it:

```
srs<-levels(as.factor(counties$Subregion))
for (i in srs) {
        that.region<-as(counties[counties@data$Subregion==i,],'SpatialPolygons')
        sr.that.region<-gUnaryUnion(that.region)
        sr.that.region<-spChFIDs(sr.that.region,i)
        if (i==srs[1]) {
                sr<-as(sr.that.region,"SpatialPolygons")
        } else {
                sr<-spRbind(sr,sr.that.region)
        }
}
```

Next, the dataframe is built and merged with the polygons, as before:

```
df<-data.frame(srs)
colnames(df)<-c("ID")
row.names(df)<-df$ID
for (i in srs) {
        df[which(df$ID==i),'Name']<-names[which(names$ID==i),"Name"]
        df[which(df$ID==i),'English']<-names[which(names$ID==i),"English"]
        df[which(df$ID==i),'Region']<-
                        counties@data[which(counties$Subregion==i)[1],"Region"]
}
subregions<-SpatialPolygonsDataFrame(sr,df)
save(subregions,file="subregions.RData")
```

Now the whole story again for the regions:

```
srs<-levels(as.factor(subregions$Region))
for (i in srs) {
        that.region<-as(subregions[subregions@data$Region==i,],'SpatialPolygons')
```

```
        sr.that.region<-gUnaryUnion(that.region)
        sr.that.region<-spChFIDs(sr.that.region,i)
        if (i ==srs[1]) {
                sr<-sr.that.region
        } else {
                sr<-spRbind(sr,sr.that.region)
        }
}
df<-data.frame(srs)
colnames(df)<-c("ID")
row.names(df)<-df$ID
for (i in srs) {
        df[which(df$ID==i),'Name']<-names[which(names$ID==i),"Name"]
        df[which(df$ID==i),'English']<-names[which(names$ID==i),"English"]
}
regions<-SpatialPolygonsDataFrame(sr,df)
save(regions,file="regions.RData")
```

For some statistics, it may prove helpful to have the data arranged by federal states. This is more or less the same as the previous aggregations, the state code being the first two digits of the county codes.

```
counties$State=substring(counties@data$ID,1,2)
srs<-levels(as.factor(counties$State))
for (i in srs) {
        that.region<-as(counties[counties@data$State==i,],'SpatialPolygons')
        sr.that.region<-gUnaryUnion(that.region)
        sr.that.region<-spChFIDs(sr.that.region,i)
        if (i ==srs[1]) {
                sr<-sr.that.region
        } else {
                sr<-spRbind(sr,sr.that.region)
        }
}
names<-read.table("States.txt",header=FALSE,
                        col.names=c("ID","Name","Qualifier","English"), sep="\t",
                        colClasses = c("character","character","character","character"),
                        fill=TRUE)
row.names(names)<-names$ID
for (i in names$ID) {
        n<-names[which(names$ID==i),"English"]
        if (n=='') {
                names[which(names$ID==i),"English"]=
                                        names[which(names$ID==i),"Name"]
        }
}
states<-SpatialPolygonsDataFrame(sr,names)
save(states,file="states.RData")
```

Helpful links

http://procomun.wordpress.com/2013/01/04/tooltips-with-r/
http://www.r-bloggers.com/maps-with-r-iii/
http://www.geodatenzentrum.de/docpdf/vg1000.pdf
http://www.r-project.org
http://www.graphviz.org
http://advsofteng.com (*ChartDirector*)

Notes

The maps displayed in this report are based on data under German federal government copyright: © GeoBasis-DE / BKG 2013 (data modified).