
CoEDL Knowledgebase

Release 1

CoEDL

Nov 29, 2022

CONTENTS

1	Auto-Chunking Audio Files into Intonational Phrases	3
2	Automatic alignment of audio and video	5
3	Limitations	9
4	Finding and correcting glossing inconsistencies	11
5	Quick automatic glossing in CLAN	15
6	Updating initials in ELAN tiers	19
7	Adding missing CV tier in ELAN	21
8	Finding and correcting spelling in CHAT files	23
9	excel2cha CLAN header generator	29
10	Introduction to Git	33
11	The CoEDL corpus platform	43
12	Quick CLAN morcode lookup	59
13	Various scripts for cleaning up the mor tier (CLAN)	63
14	Set up Nectar	71

This is a collection of scripts that have been written for working with language data.
Better formatting coming soon...

AUTO-CHUNKING AUDIO FILES INTO INTONATIONAL PHRASES

author: T. Mark Ellison date: 2017-05-01

tags: - Segmentation - Intonational Phrase - Silence - PRAAT

categories: - Tips

1.1 Introduction

[Eri Kashima](#) and I have found a neat way of chunking speech from the audio file, as a first step in transcription. Initial efforts using silence-detection in ELAN were not successful. Instead, we found that PRAAT's silence detection did the job quite well once the right parameters were chosen.

We use PRAAT's **Annotate >> To TextGrid (silences)...** option from the PRAAT file window. This option is available once you have loaded the *.wav* file. Our parameter settings are:

- **Minimum pitch** 70Hz
- **Silence threshold (dB):** -35
- **Minimum silent interval duration(s):** 0.25
- **Minimum sounding interval duration(s):** 0.1
- **Silent interval label:** (empty)
- **Sounding interval label:** ***

A detailed walkthrough - of chunking by PRAAT for a file normally explored in ELAN - can be seen on [Eri's blog page](#) on the topic.

AUTOMATIC ALIGNMENT OF AUDIO AND VIDEO

author: Sasha Wilmoth, Ola Olsson date: 2017-05-09

tags: - CLAN - ELAN - Python - Docker

categories: - Scripts - Tutorials

2.1 Introduction

`av_align.py` is a Python script designed to solve the problem of unaligned audio and video recordings of the same session. It calculates the most likely offset between two media files, and pads them accordingly. It also adjusts the timestamps of an ELAN or CLAN transcript if necessary.

The script runs on Windows/Mac/Linux. We've created a Docker¹ container so there's minimal setup.

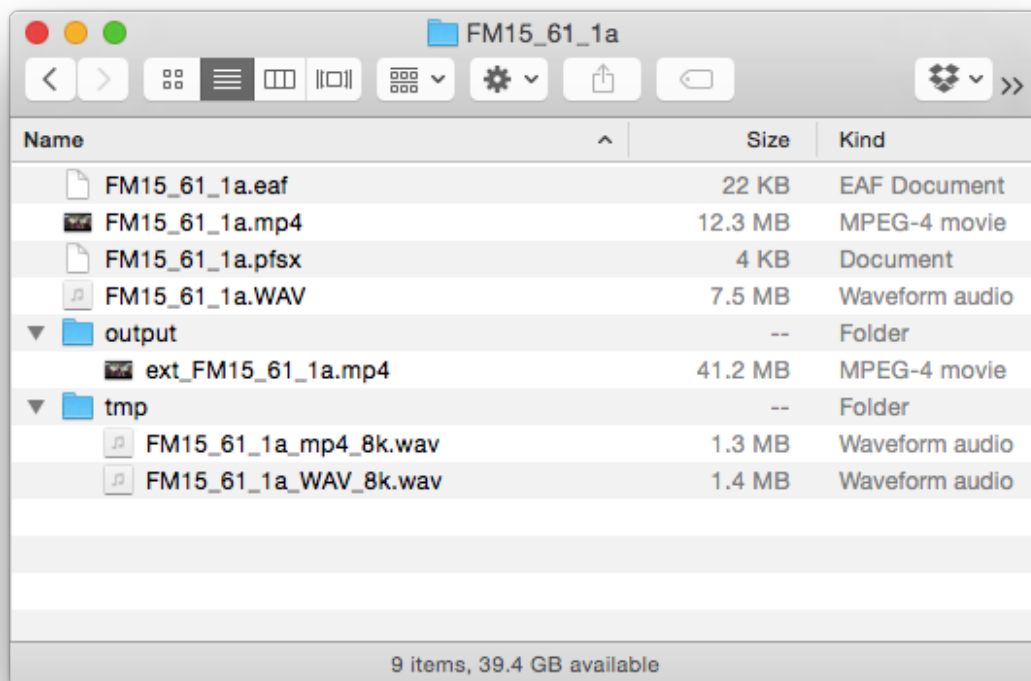
2.1.1 Input

The input is a directory containing:

- Video: mov or mp4
- Audio: wav or mp3
- Transcript: eaf or cha
 - If there's no transcript, the media files will simply be padded to match each other.

2.1.2 Output

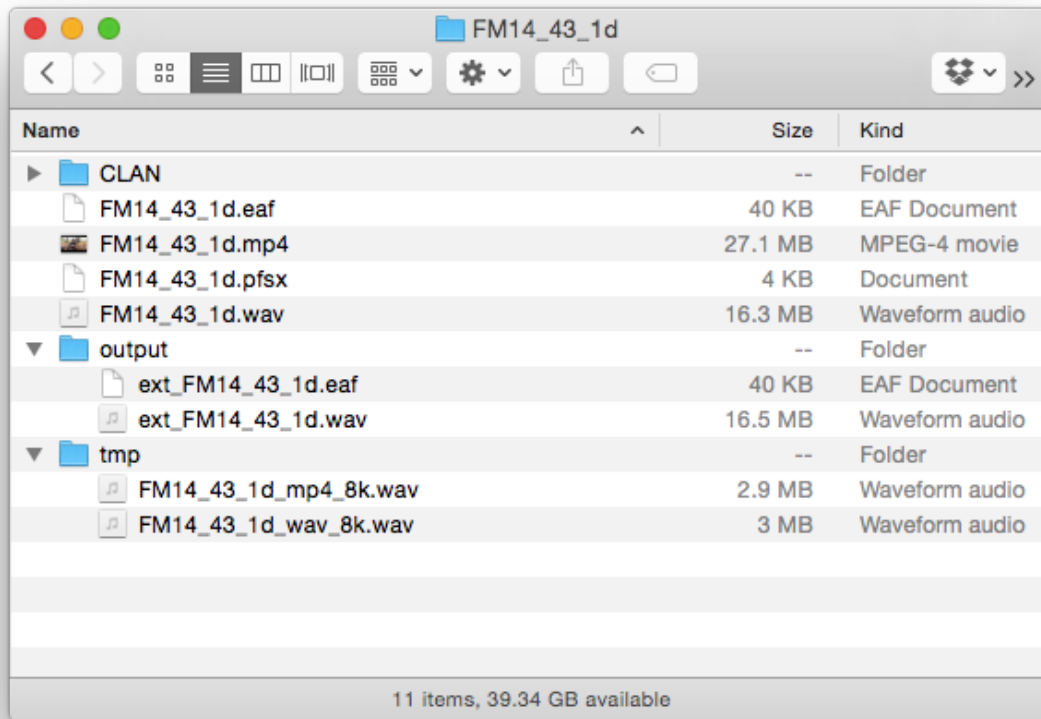
If the audio starts before the video, the script will output an extended version of the video, with grey frames at the beginning. A temporary folder is also created containing the audio files used by the script.



Example

of script output - extended video

If the video starts before the audio, the script will output an extended version of the audio, with silence at the beginning, as well as a time-adjusted transcript. **The transcript is assumed to be time-aligned to the audio file.**



Example

of script output - extended audio

2.1.3 Options

You can run the script with the following options:

- [-h] Help message
- [-t TMPDIR] Location of temporary audio files. Default is ./tmp.
- [-d INPUTDIR] Input directory. Default is . (current directory).
- [-f FFMPEGPATH] av_align.py uses [FFmpeg](#) to extract and pad the media files. FFmpeg is included in the Docker container, so this option isn't necessary unless you're running av_align.py on your own system, and FFmpeg is not on your system path.
- [-v] Verbose - for troubleshooting.

2.2 Instructions

1. Download and install Docker [here](#).
2. Move to the directory with your data and run the script:

```
cd /path/to/your/corpus/  
docker run --rm -v $(pwd):/to_align coedl/av_align python /av_align.py -d /to_align/  
↪Session001
```

`$(pwd)` means present working directory, `/to_align` is what your corpus will be called when mounted onto the image.

You can see these steps in action here:

```
bash-3.2$ tree  
.  
├── FM041_B_compressed  
│   ├── FM041_B.cha  
│   ├── FM041_B.mp3  
│   └── FM041_B.mp4  
  
1 directory, 3 files  
bash-3.2$ docker pull coedl/av_align  
Using default tag: latest  
latest: Pulling from coedl/av_align  
d54efb8db41d: Already exists  
f8b845f45a87: Already exists  
e8db7bf7c39f: Already exists  
9654c40e9079: Already exists  
6d9ef359eaaa: Already exists  
a3ed95caeb02: Already exists  
f9d05c3c96e7: Already exists  
0befd19b3f19: Already exists  
dcd8b0a4dfb3: Pull complete  
83538c03f3d2: Downloading 137.6 MB/218.3 MB  
d817f4e33826: Download complete  
0cfe081578a2: Download complete  
b12615374f45: Download complete
```

asciicast

If you're not a techy person and need some help setting up your workflow, feel free to [email me](#).

LIMITATIONS

- `av_align.py` does not take into account **time stretch**. If one of your media files is stretched relative to the other, it will find an average offset.
- At this stage, the script assumes you only have **one audio file and one video file**. It doesn't work if you're comparing audio + audio, video + video, or more than two files.
- When adjusting timestamps, it is assumed that these are in milliseconds. If you have set your annotations in ELAN to align with frames, proceed with caution.
- The script loads both wave forms into memory. This shouldn't be a problem unless you're working with very long recordings (hours).

3.1 Acknowledgements

`av_align.py` was written by Ola Olsson and tested by Sasha Wilmoth, on data collected by Felicity Meakins. Thanks to Nay San for setting up the Docker image.

¹ Wait, what's Docker? If you need a metaphor: the Docker container (or image) is like a tiny computer that only runs `av_align.py`, and you're plugging in your corpus like a USB stick.

FINDING AND CORRECTING GLOSSING INCONSISTENCIES

author: Sasha Wilmoth date: 2017-05-26

tags: - Glossing - Flex - Python - Appen

categories: - Tutorials

4.1 Introduction

The analysis of a language's grammar can evolve over time. So too can glossing preferences. How can one ensure that an entire corpus is glossed consistently, as the lexicon is continually updated?

This post introduces a simple process for finding glossing inconsistencies in flextexts, and then automatically making changes across a corpus.

1. `inconsistentGlosses.py` finds glossing inconsistencies, and compares them to the most up-to-date lexicon.
2. The user manually corrects the output to reflect the preferred gloss.
3. `updateGlosses.py` automatically applies these changes across a corpus.

This process is designed to be iterative, with a bit of back and forth between running `inconsistentGlosses.py`, updating the lexicon, and fixing individual examples in context.

4.2 Requirements

`inconsistentGlosses.py` and `updateGlosses.py` require Python 2.7, and do not currently work with Python 3. They work on Mac, and have not been tested on Windows.

Both scripts can be found [here](#).

4.3 `inconsistentGlosses.py`

4.3.1 Input

`inconsistentGlosses.py` takes two arguments: a Flex lexicon, and a directory containing `.flextext` files. It looks in all subdirectories.

The command is:

```
inconsistentGlosses.py [-h] YourFlexLexicon /path/to/your/corpus/ [-v] > output [2>
↳ErrorReport]
```

-h is a help message, -v is a verbose option that prints which files the invalid glosses were found in.

4.3.2 Output

The script looks for morphs with glosses that aren't in the lexicon, and outputs a table which can be copied into a spreadsheet. This is the non-verbose output:

```
Morph | Original gloss | Gloss to correct | Frequency | In lexicon | Comments |
18 | | Consider: 3PAUC.DO | | | | -wanku | COM | COM | 32 | ✓ | | also | also | 1 | | Consider: COM | | | | Dawun |
MISSING GLOSS | MISSING GLOSS | 3 | | Variant of dawun. There is no gloss for Dawun or dawun in the lexicon. |
| | | batbat | right | right | 2 | | There is no gloss for batbat in the lexicon. | | | | batj | bring | bring | 74 | ✓ | | watch | watch |
14 | ✓ | | cook | cook | 2 | ✓ | | take/bring | take/bring | 1 | | Consider: cook, bring, have.something.caught.in.one's.throat,
watch, submerge ... |
```

The script also reports some stats to your terminal, as well as any morphs it finds that aren't in the lexicon. You can save this with `2> ErrorReport.txt` if you want to. For the Murrinhpatha data, the error report looks like this (if the verbose option is switched on, there's another column with file names):

118 flextext files analysed

95 morphs with inconsistent glosses

15 morphs missing from lexicon

```
Morph not in lexicon | Gloss in text | Frequency | Comments |
-dhangu | source | 4 | -n | 3PL.DO | 10 | Allophone:
-pun is citation form -pirra | 3PL.IO | 5 | Allophone: -wirra is citation form -rru | 3PAUC.IO | 6 | Allophone: -pirru is
citation form -wayida | reason | 1 | =ka | =TOP | 35 | ... |
```

We can then add all the missing morphs to the lexicon (or correct errors), add glosses to the lexicon where they were missing, and check glosses in context where necessary. If we run the script again, the output will be a bit smaller (e.g. *batbat* no longer shows up because we've added 'right' as the gloss in the lexicon). Then we can correct the third column like so:

```
Morph | Original gloss | Gloss to correct | Frequency | In lexicon | Comments |
18 | | Consider: 3PAUC.DO | | | | -wanku | COM | COM | 32 | ✓ | | also | COM | 1 | | Consider: COM | | | | Dawun |
MISSING GLOSS | Darwin | 3 | | Variant of dawun. Consider: Darwin. | | | | batj | bring | bring | 74 | ✓ | | watch | watch |
14 | ✓ | | cook | cook | 2 | ✓ | | take/bring | bring | 1 | | Consider: cook, bring, have.something.caught.in.one's.throat,
watch, submerge ... |
```

This file becomes the input for `updateGlosses.py`, so we can make these changes across all the files automatically.

4.4 updateGlosses.py

To update your lexicon:

```
updateGlosses.py [-h] CorrectionsFile /path/to/your/corpus/
```

The script creates a new directory called `UpdatedFlextexts` in your current working directory, and makes new copies of any flextext files containing incorrect glosses.

Note: Flex might object when you try and open the new files. If this happens, make sure the new glosses match the latest version of the lexicon. There may be an occasional problem when a gloss was missing in the original file.

4.5 Acknowledgements

Thanks to Rachel Nordlinger and John Mansfield for the Murrinhpatha data, and Jason Johnston (with the support of Appen) for Python help.

QUICK AUTOMATIC GLOSSING IN CLAN

author: Sasha Wilmoth, Simon Hammond, Hannah Sarvasy date: 2017-05-30

tags: - Glossing - CLAN - Python - Appen

categories: - Scripts

5.1 Introduction

Hannah Sarvasy is conducting a longitudinal study of five children acquiring the Papuan language Nungon in a remote village of Papua New Guinea. This entails monthly recordings of the target children that are conducted and transcribed by four Nungon-speaking assistants in the village. The corpus will total 104 hours of transcribed natural speech recordings. Hannah's task is to add interlinear morpheme-by-morpheme parses and glossing and free English translations to the Nungon transcriptions, which are in the CHAT format. The glossing is to be done

CLAN's built-in MOR program is an effective tool for glossing if you've already transcribed your data with morpheme boundaries, and you have a lexicon in .cut format.

However, Hannah had some slightly different needs, and I figured we could write a simple script that could correct spelling on the transcription tier, add a pre-determined parse, and add a gloss.

This script, `glossFile.py` is not an automatic parser - it simply takes the words you feed it, and makes no guesses about their structure.

It might be useful for you if you want to do a rough automatic pass before checking through your glossing manually, and you don't already have a large lexicon or parser set up. It was also designed to suit our particular circumstances, as the data had a lot of variation in spelling and spacing, and many repeated words (so we could cover a lot of ground without a parser).

5.2 Instructions

5.3 Requirements

`glossFile.py` can be found here [here](#). You will need a GitLab account to request access.

`glossFile.py` requires Python 2.7, and does not currently work with Python 3. It works on Mac, and has not been tested on Windows.

5.4 Step 1: Create a lexicon

Compile a list of words in your data, sorted by frequency. You can use CLAN's `FREQ` command to do this.

I also made a pre-populated lexicon based on some files that had already been manually glossed.

I then sent all the remaining words to Hannah, sorted by frequency, in the following format:

```
Word | Spelling Correction | %gls | %xcod | Frequency —|—|—|—|—| no | no | no | 1500 ma | ma | ma | 772 oro | oro
| oro | 752 dek | dek | dek | 543 ho | ho | ho | 520 diyogo | diyogo | diyogo | 230 nano | nano | nano | 200 awa | awa |
awa | 187 hewam | hewam | hewam | 175 arap | arap | arap | 174 Lisa | Lisa | Lisa | 169 dec | dec | dec | 167 bop |
bop | bop | 165 gon | gon | gon | 162 to | to | to | 155 mai | mai | mai | 151
```

Hannah then checked through this list, and for each entry:

1. Made a correction in the second column if necessary
 - In order to reduce the manual effort of entering glossing for misspelled words multiple times, the third and fourth columns are ignored if there is a spelling correction.
 - Corrected words are then glossed according to the correctly-spelled entry (which already existed in most cases).
2. Added the appropriate morpheme boundaries
 - This example shows the most frequent items, which happened to include a lot of suffixes and clitics. If a `-` or `=` is added in the third column, the transcription tier is corrected to reflect this, and a space is removed.
3. Added a gloss for the whole word.

Note:

- Multiple entries for homophones are fine, and will be separated with a `#` symbol in both the `%gls` and `%xcod` tiers for manual disambiguation.
- If a word is not found in the lexicon, it is just printed as is on all tiers in the output.
- If a 'word' is not a morpheme at all, but should just be connected to the previous word, a full stop can be added in the third column. E.g. `.kak` tells the script that the word `kak` is actually a part of the previous word.

```
Word | Spelling Correction | %gls | %xcod | Comments —|—|—|—|—| no | no | -no | -3sg.poss|This means that example
no will become exampleno/example-no/example-3sg.poss on the three tiers. ma | ma | =ma | =rel| oro | oro | oro |
adv^well| dek | dek | =dek | =loc| ho | ho | =ho | =foc| diyogo | diyogo | diyogo | q^how| nano | nanno | nano | |The script
only looks at the second column and ignores the rest. awa | aawa | awa | | hewam | hewam | hewam | n^tree.kangaroo|
arap | arap | arap | n^mammal| Lisa | Lisa | Lisa | n^name| dec | dec | =dec | | bop | boop | bop | | gon | gon | =gon | =restr|
to | to | =to | =foc| mai | mai | =ma-i | =rel-top|
```

5.5 Step 2: Run the script

Once you have a 4-column lexicon file (we can discard the frequency column), the command is:

```
python glossFile.py [-h] lexicon input.cha > output.cha
```

This will turn a passage like this (from `09_septemba_niumen.cha`):

```
*CHI:      biksa yandinga itdok.
%com:      dogu digu yandinga itdok.
*MOT:      opmu menon ngo nungon to duwonga itdok ngo.
```

(continues on next page)

(continued from previous page)

*MOT:	ngo ngo.
*CHI:	nnn nandumau.

into this:

*CHI:	biksa yandinga itdok.
%gls:	biksa yandinga it-do-k
%xcod:	tpn^picture yandinga v^be-rp-3sg
%eng:	
%com:	dogu digu yandinga itdok.
*MOT:	opmu menon ngo nungonto duwonga itdok ngo.
%gls:	opmou menon ngo nungon=to duwo-nga it-do-k ngo
%xcod:	adj^small menon d^prox q^what=foc v^sleep-mv.ss v^be-rp-3sg d^prox
%eng:	
*MOT:	ngo ngo.
%gls:	ngo ngo
%xcod:	d^prox d^prox
%eng:	
*CHI:	nnn nandumau.
%gls:	nn nandu=ma au#nandu=ma-u
%xcod:	ij n^non.specific=rel adj^other#n^non.specific=rel-top
%eng:	

Note:

- *oe ho* is corrected to *oeho*,
- *yandinga* and *menon* are just printed as is (they weren't in the lexicon),
- *nandumau* has a couple of options to choose from, *nandu=ma au* (n^non.specific=rel adj^other) or *nandu=ma-u* (n^non.specific=rel-top)

5.6 Acknowledgements

glossFile.py was written by Simon Hammond thanks to Appen's partnership with CoEDL.

All Nungon data shown here was collected by Hannah Sarvasy, and transcribed by Nungon-speaking assistants.

UPDATING INITIALS IN ELAN TIERS

author: Sasha Wilmoth date: 2017-06-01

tags: - Elan - Python

categories: - Scripts - Tutorials

6.1 Introduction

If you use speaker's initials in tier names in ELAN, you might need to update them occasionally. You can do this manually in ELAN by clicking Tier > Change Tier Attributes, and then typing in the new initials for every single tier.

If that's too time-consuming, you can use **changeSpeakerInitials.py**. This is a simple Python script I wrote for Ruth Singer. Ruth had multiple participants in her corpus with the same initials - let's call them Mickey Mouse (MM) and Minnie Mouse (MM).

The tiers look something like this: 

6.2 changeSpeakerInitials.py

6.2.1 Requirements

changeSpeakerInitials.py works with Python 2.7. It works on Mac, and has not been tested on Windows.

The script can be found [here](#).

6.2.2 Instructions

Instead of manually changing each tier in each file to MM1 and MM2, we can make a two-column (tab-delimited) file like this:

New Initials | Name — | — MM1 | Mickey Mouse MM2 | Minnie Mouse

The command is:

```
changeSpeakerInitials.py [-h] Speaker_initials.txt /path/to/your/corpus
```

The script looks at who the participants are in each file, and changes the initials in the tier name according to the speaker database. So 'rf@MM' becomes either 'rf@MM1' or 'rf@MM2'.

Any updated ELAN files are put into a new directory called UpdatedInitials.

The script also prints an error message if it finds any names not in the speaker/initial file, as well as the file(s) that each name was found in.

Note: The script finds the initials to replace by looking for capital letters or numbers following an @ symbol in the tier name.

ADDING MISSING CV TIER IN ELAN

author: Sasha Wilmoth date: 2017-08-24

tags: - Elan - Python

categories: - Tutorials

7.1 Introduction

Until the recent release of [ELAN 5.0](#), it was not possible to automatically parse and gloss transcriptions in ELAN. A commonly used workaround is to export from ELAN into Toolbox, and import the interlinearised texts back into ELAN. In the process, some tiers might be altered.

Ruth Singer sent me some files that had all been nicely glossed, but, as you can see in the screenshot below, the utterance has been tokenised into single words in the XV tier.

![Screenshot of ELAN file with no CV tier](Screen Shot 2017-08-24 at 4.27.06 pm.png)

I wrote a script in Python that adds the original utterance back in for each participant, based on the words in the XV tier. The output looks like this:

![Screenshot of ELAN file with CV added back in](Screen Shot 2017-08-24 at 4.27.30 pm.png)

7.2 addCVtier.py

7.2.1 Requirements

addCVtier.py requires Python 2.x. It works on a Mac, and has not been tested on Windows.

The script can be found [here](#).

7.2.2 Running the script

The script runs on a single .eaf file at a time, and outputs a new file with the extension _CV.eaf. The command is simply:

```
addCVtier.py File001.eaf
```


FINDING AND CORRECTING SPELLING IN CHAT FILES

author: Sasha Wilmoth date: 2017-08-29

tags: - Tutorial - Spelling standardisation - CLAN - CHAT - Gurindji Kriol - Python

categories: - Scripts

8.1 Introduction

The Gurindji Kriol corpus is glossed using the MOR command in CLAN, which looks up each token in the transcription tier, and adds a mor-code from the lexicon accordingly. In order to have a squeaky clean mor-tier, we need to ensure that each token is accounted for, and there are no typos in the transcription or new words to be added to the lexicon. At this stage, we can also fix any known misanalyses of common words or sequences (for example, *dat tu* should be *dat _tu*, as *_tu* is a dual suffix and not a numeral in this context).

I have written two scripts to automate this process: one finds unknown tokens in the transcription tier and outputs a two-column file, the other corrects all CHAT files in a particular directory according to said two-column file.

The two-column checked words file is **cumulative** - once you correct a typo once, you should keep this file and use this valuable information next time you're coding new data.

The process is also **iterative**. You should always double check the corpus with the first script after running the second, to make sure you've caught everything.

8.2 Instructions

8.3 Requirements

Both scripts require Python 2.x. They work on Mac and have not been tested on Windows.

The scripts can be found [here](#).

8.4 findUnknownTokens.py

8.4.1 Input

Lexicon(s)

The script allows you to have more than one lexicon - we use two for Gurindji Kriol, as it is a mixed language. They look like this:

```
@UTF8
_a {[scat case:loc]} "_ta&g" =LOC=
_ja {[scat case:loc]} "_ta&g" =LOC=
_jawung {[scat der:having]} "_jawung&g" =PROP=
_ji {[scat case:erg]} "_tu&g" =ERG=
_jirri {[scat case:all]} "_jirri&g" =ALL=
_ju {[scat case:erg]} "_tu&g" =ERG=
...
```

```
...
yutu {[scat pro]} "yundubala&2DU&k" =you_two=
yutubala {[scat pro]} "yundubala&2DU&k" =you_two=
yuu {[scat interj]} "yu&k" =yes=
yuwai {[scat interj]} "yuwai&k" =yes=
zebra {[scat n:animal]} "zebra&k" =zebra=
```

Don't worry if your syntax is slightly different - the script only looks at everything before the first space.

Checked words file

I started with a basic 'checked words' file that looked like this:

```
| Error | Correction | |—|—| | dat tu | dat _tu | | boy | boi | | shoulda | sholda | | kangkaru | kengkaru |
```

This is a tab-delimited plaintext file, with incorrect strings in the first column, and corrections in the second column. The strings can contain **up to 2 tokens**. (Let me know if you would like support for longer strings)

I'm using a checked words file because I know about these errors already, and I don't need the script to tell me about them again. You can also run the script without a checked words file. This is handy if you're checking your data for the first time, or if you're doing a final check after correcting everything (and all of these errors should be cleaned up already).

Directory

findUnknownTokens.py will look at any file ending with .cha, in all subdirectories within the given directory. The Gurindji Kriol corpus is structured like so:

![Screenshot of Gurindji Kriol directory structure](Screen Shot 2017-08-29 at 2.25.01 pm.png)

The script only looks on the transcription tier, which begins with an asterisk, and may carry on over the next line, like this:

![Screenshot of Gurindji Kriol directory structure](Screen Shot 2017-08-29 at 2.39.30 pm.png)

8.4.2 Command

With those inputs, the command I use is:

```
findUnknownTokens.py -l lex_gurindji.cut lex_kriolgen.cut -c checkedwords.txt -d /path/
↳to/Transcription/folder > output.txt
```

The different arguments are:

```
-h      Help
        If you use this option, the script doesn't run, it just prints a short help_
↳message
-l      Lexicon
        At least one lexicon must be provided
-c      Checked words file
        This is an optional argument, if you would like to ignore previously-identified_
↳errors in the output file. No more than one checked words file can be provided.
-d      Directory
        The path to the directory containing .cha files
```

8.4.3 Output

Using the above command gives me an output file with over 200 unidentified tokens. It looks something like this:

```
| Unknown token | For correction| |—|—| | _rayinyj | _rayinyj | | ppuntanup | ppuntanup | | -aran | -aran | | xx | xx | |
| playing | playing | | footy | footy | | kajirrii | kajirrii | | lajamanu | lajamanu | | writing | writing | | _waija | _waija | | ayglas
| ayglas | | _bat_karra | _bat_karra | | wood | wood | | marlaku | marlaku | | ... |
```

At the bottom of the file, it also says:

=== The following tokens are already in the 'checked words' file.===

===They will be corrected if you run correctCHATSpelling.py. ===

```
| Token | Correction | |—|—| | kangkaru | kengkaru | | shoulda | sholda |
```

When we go through this file, we have three options:

- For tokens we want to correct automatically, we can do so in the second column and include it in the checked words file.
- We might need to check some things in context. If we fix them all, we should remove this line from the checked words file.
- There might be some new words to add to the lexicon. We should do so, and then remove them from the checked words file.

For example:

```
| Unknown token | For correction| |—|—|—| | _rayinyj | _ranyj | | ppuntanup | puntanup | | -aran | _aran | | xx | xxx | |
| playing | plei_ing | | ~footy~ | ~footy~ | (This wasn't in the lexicon, so I'll add it) | | kajirrii | kajirri | | lajamanu
| Lajamanu | | writing | rait_ing | | _waija | _walija | | ayglas | eyeglas | | _bat_karra | _bat_karra | | wood | wud | |
~marlaku~ | ~marlaku~ | (This could be warlaku 'dog' or marluka 'man' - I'll have to check them in context and
remove this line from the checked words file) | | ... |
```

When I've gone through the full list, I'll add these to my previous checked words file (the one with *boy* and *kangkaru*). This then becomes the input for the next script, and is very valuable information to have next time you're cleaning up your corpus.

Before you proceed, I would recommend running the script again with the updated checked words file and lexicon, to see if there's anything you missed.

8.5 correctCHATSpelling.py

This script only needs a checked words file and a directory containing .cha files. The command is:

```
correctCHATSpelling.py -c checked-words-file -d directory
```

You don't need to specify the output, as it makes new files in the same directory as the original files, with the extension .cha.corrected. Check through your files and make sure things have been corrected properly. When you're ready, you can rename the corrected files with a command like this (be warned: this will replace your original files if you haven't made a copy elsewhere):

```
for file in */*/*.cha.corrected; do mv "$file" "`basename "$file" .cha.corrected`.cha";  
done
```

This is looking for the corrected files in two nested subdirectories, and would be run from the 'Transcription' directory in the above screenshot.

8.6 Conventions

Please note that these scripts were developed for the Gurindji Kriol corpus, and as such take into account some specific conventions that may not apply to your data.

findUnknownTokens.py will not report some tokens:

- Commas, full stops, exclamation marks, and question marks are not reported in the output. The direct speech marker +” is also not reported.
- Commas are stripped from the tokens before looking them up in the lexicon.
- Tokens beginning with capital letters [A-Z] are ignored, as these are automatically coded as proper nouns and do not need to be in the lexicon.
- Tokens beginning with an ampersand are ignored, as these are foreign words.
- xxx (representing unintelligible speech) is ignored.

correctCHATSpelling.py will:

- Add a space before a full stop, exclamation mark and question mark, if one doesn't already exist.
- Remove a space before a comma.

8.7 Notes

All the Gurindji Kriol data shown has been recorded by Felicity Meakins and Cassandra Algy.

If you need help with this process or would like to request any changes (such as support for ELAN), please [get in touch](#) and I'll be happy to help.

EXCEL2CHA CLAN HEADER GENERATOR

author: Jonathon Taufatofua date: 2017-09-19

tags: - CLAN - Excel - metadata

categories: - Scripts

Metadata is “data about data”. Recording metadata is important for linguistic field work. When documenting language, the metadata contains information such as location, time, participant information, recording format, media filenames, etc. Metadata can provide information about the social context of a documentation activity and provide a locator for a resource once it enters a larger collection.

However, it can be a burden to create and use metadata! There are many ‘standard’ forms of metadata, many ways to create and store metadata, and each linguist has their own preferred method.

Some transcription software requires particular metadata to associate the transcription with a media file.

This script was developed to copy data from a spreadsheet into the header fields of a CLAN transcription, so that a linguist wouldn’t need to input the data twice. This small step is significant in reducing the amount of work required when beginning transcribing.

9.1 Download

Download the script from github.com/CoEDL/clan-helpers

9.2 Screenshots

Example spreadsheet with metadata:

Corpus name:																
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
Corpus name	Length of audio	Date recorded (DD-MM-YYYY)	Location info	Speakers	Language	Investigator	Activity	Related Files	Comments on Audio	Related Sessions	Transcribed	Transcribed by	Checked	Checked with		
1	JD_63BC_01	3:14min	11-Nov-2011	Rome	MCI, MAN	lat	JDO	>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed sed sollicitudin sem, in tempor erat. Vestibulum gravida luctus tortor, non posuere libero vulputate ac.			N	JDO	N			
2	JD_63BC_02	15:26min	11-Nov-2011	Rome	MCI, MAN	lat	JDO	Mauris mauris lectus, posuere non tempus eu, aliquet at enim. Curabitur enim augue, elementum non pellentesque tempor, eleifend vel enim.			N	JDO	N			
3	JD_63BC_03	53:58min	11-Nov-2011	Rome	MCI, MAN	lat	JDO	Donec sem ligula, commodo non odio et, fringilla posuere diam. Integer sodales vulputate odio, quis consectetur eros volutpat vel.			N	JDO	N			
4	JD_63BC_04	32:38min	11-Nov-2011	Rome	MCI, MAN	lat	JDO	Ut lacina commodo justo id commodo. Praesent nec placerat dolor.			N	JDO	N			
5	JD_63BC_05	46:26min	11-Nov-2011	Rome	MCI, MAN	lat	JDO	Etiā eu erat sit amet nulla posuere congue eu id mauris. Proin vitae risus nunc.			N	JDO	N			
6																
7																
8																
9																
10																
11																
12																
13																
14																
15																

Ready

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
Participant surname	Participant first name	CLAN code	Age (2011)	Sex	Languages (order of proficiency)	Language Comments	Relationships with other participants	Usual role in recordings								
1	Cicero	Marcus	MCI	2117	M	lat, grk, etc		speaker								
2	Antonius	Marcus	MAN	2094	M	lat, grk, egp		speaker								
3	Doe	Jane	JDO	21	F	lat	Does not like Cicero	investigator								
4																
5																
6																
7																
8																
9																
10																
11																
12																
13																
14																

Excel

metadata spreadsheet

And here's how it ends up in CLAN:

JD_63BC_01.cha																
1	@Begin															
2	@Languages: lat															
3	@Participants: MCI MarcusCicero Speaker, MAN MarcusAntonius Speaker,															
4	JDO JaneDoe Investigator															
5	@ID: lat, grk, etc mycorpus MCI M Speaker															
6	@ID: lat, grk, egp mycorpus MAN M Speaker															
7	@ID: lat mycorpus JDO F Investigator															
8	@Media: JD_63BC_01, audio															
9	@Location: Rome															
10	@Transcriber: JDO															
11	@Comment: Lorem ipsum dolor sit amet, consectetur adipiscing															
12	elit. Sed sed sollicitudin sem, in tempor erat. Vestibulum gravida															
13	luctus tortor, non posuere libero vulputate ac.															
14	@Date: 11-NOV-2011															
15																

CLAN

header

9.3 Instructions

The script requires the session information (time/location/participants...etc) in one Excel sheet as well as a second sheet in the same workbook, listing the participant IDs for CLAN. It relies on a particular format of the data (they must be separable into CLAN header-like columns so its just pick and place rather than chopping and changing). Currently it needs an exact column ordering but it could be modified for a more general case.

More steps coming soon...

INTRODUCTION TO GIT

author: Ben Foley, Nay San date: 2017-09-23 updated: 2017-10-06

tags: - Git - Version Control

categories: - Tutorials

This is a beginner's guide to using Git version control with the SourceTree program.

10.1 Contents

- *What is git?*
- *Why use version control?*
- *What does it look like?*
- *How do we use git?*
- *Real-life linguist workflows*
- *Hands on*
 - *Getting set up*
 - *Software*
 - *A sandbox*
 - *First commit*
 - *History*
 - *Let's branch*
 - *Remotes*
 - *Protecting branches*
 - *Dealing with merge conflicts*
- *Definitions/glossary*
 - *More definitions*
- *Further reading*
 - *Stay tuned for more git workshops*

10.2 What is git?

Git is a *version control* tool, a way to keep track of changes to files.

Think about MS Word's Track Changes, but it's so much better. It's also better than copying files into folders with dates (or folder names like "final", "final2", "last-changes").

10.3 Why use version control?

Version control documents your changes to files, storing the date, time and author info as a "commit". When you save your changes you can describe the changes you made with a message. This version history is a valuable log, a reference which you can look back through to find the point at which data changed.

Version control gives you certainty about the state of your data, especially when collaborating. Using Git, you can compare the changes you make to your copy of data against what others have done to theirs.

When your changes have been committed, they stay in that state (you can come back to that point if you ever need and they will not have altered).

It's a great way to work collaboratively. Many people can work on a project, each person can have a local copy of the project and work on their local version without affecting other people's copies. This also makes it useful for offline work. If you have a copy of a Git project on your computer, no network is needed to access a history of the project's changes because all the history is stored within the project. You can edit the project while offline and synchronise later when you have online access.

10.4 What does it look like?



A simple tree (or graph of commits) looks like this:

```
[c]
|
[b]
|
[a]
```

A slightly more complex graph has branches:




```
    [f]
    |
[e]  |
  \  |
    [d]
    | [c]
    | /
    [b]
    |
    [a]
```

Here's how graphs look in SourceTree (a visual Git editor).

Graph	Description	Commit	Author	Date
	 master Add frog	7890a29	Ben Foley <ben@...>	Today, 10:11 AM
	Add dog	d4cb599	Ben Foley <ben@c...>	Today, 10:10 AM
	Add cat	a91c711	Ben Foley <ben@c...>	Today, 10:10 AM

A

simple graph

Graph	Description	Commit	Author	Date
	 master Add pony	4bb80ac	Ben Foley <ben@...>	Today, 10:29 AM
	 show-prep Add todo list for show	0c81086	Ben Foley <ben@c...>	Today, 10:28 AM
	Added frog	7890a29	Ben Foley <ben@c...>	Today, 10:11 AM
	Added dog	d4cb599	Ben Foley <ben@c...>	Today, 10:10 AM
	Added cat	a91c711	Ben Foley <ben@c...>	Today, 10:10 AM

A

graph with a branch

Git commands either add nodes to the graph or navigate through the graph. The graph of commits within a project is called a repository.

Git repositories can be stored on your local computer and remotely on a server like github.com, gitlab.com or bitbucket.com.

10.5 How do we use git?

The basic Git workflow goes something like this:

- **Pull** files from a remote repository.
- Work on the files locally.
- **Add** the changed files to a staging area.
- **Commit** the changes, which stores the changes.
- **Push** that commit to a remote repository.

We'll do these soon...

10.6 Real-life linguist workflows

For a linguist working solo, a typical project might contain one remote repository, cloned to their computer. In this setup, it is common to only have one branch.

When collaborating, the setup will change depending on the degree of trust within the group.

For small numbers of trusting collaborators, the remote repository should have a “protected master branch” which prevents people from pushing directly to that remote branch. Each approved contributor can clone the project to their computer and work on their own branch. When they want to share their work, a “pull request” or “merge request” is made to bring the changes from their branch into the remote’s master branch. The request can check whether there are conflicts in what has been changed which need to be resolved before merging happens. This helps with avoiding breaking other people’s work.

For larger (especially public) groups, the remote repository is forked by each contributor. They then clone their remote locally, work locally and push back to their remote. Pull/merge requests can still be made from their forked repository to the original repository, to bring their work back into the original.

10.7 Hands on

10.7.1 Getting set up

Software

For the rest of this guide, we'll use SourceTree to work with the Git repository, and VS Code to edit files.

1. Install SourceTree from <https://www.sourcetreeapp.com/>.
 - 1.1 Click the download button.
 - 1.2. Open the downloaded file (EXE for Windows or DMG for Mac).
 - 1.3. You need an Atlassian account to use SourceTree. If you have one already. click *Use an existing account*. To create an account, click *Go to My Atlassian* and follow the prompts to create a new account.
 - 1.4. If/when SourceTree says that it cannot find an installation of Git, click the option to install an **Embedded version of Git**. We don't need Mercurial.
 - 1.5. Open SourceTree and go to Preferences. Enable the option to *Allow SourceTree to modify your global configuration files*. Then add your name and email address in the Default user information fields. Note that this information will be publicly available if you push commits to a public remote repository.
2. Get VS Code from <https://code.visualstudio.com/>
 - If you do not have administrator/install access on your Windows machine, you should get the .zip file version of VS Code from <https://code.visualstudio.com/docs/?dv=winzip>

A sandbox


1. For today's play, make a folder somewhere easy to access, preferably without spaces in the path. For example (on mac), ~/Documents/learn-git
2. Open SourceTree and go to **File > New/Clone > Add Existing Local Repository**.
 - 2.1. Browse to the folder you just created.
 - 2.2. SourceTree should prompt us to create a local repository. Click **Create**. This just initialised git version control for this folder!
 - 2.3. Double click the icon in the SourceTree repo list to open the main window for this repo. Nothing to see here yet though. Let's add some files.
3. Open the folder in Visual Studio. **File > Open** then select your learn-git folder.

10.7.2 First commit

1. Let's add a file in the folder. Using VS Code, **File > New File** then name the new file **wordlist**. The file will open in the editor pane when it has been created. Type some words into the new file and save it.
2. Now, back to SourceTree. In the sidebar **Workspace > File** status we should see our new file. Git sees that the file is here but the change hasn't been staged yet. Over on the right-hand side, we see a "diff" of the file changes.
3. Tick the checkbox next the filename in the **Unstaged files** area. This adds the file changes to the staging area, ready for a commit.
4. Now, compose a short, descriptive message in the Commit message box, something like **Add first words to wordlist**. And press the **Commit** button.
5. Pat yourself on the back, your first change has been committed :-)
6. Let's repeat that, so we can see how **history** and **diff** work.
7. Add some more words to the wordlist file. Save in VS Code, then stage (add) and commit in SourceTree.
8. Delete a word or two, save, add and commit.

10.7.3 History

Now we have three commits in the repository. Looking at **Workspace > History** we can see a log of our changes.

Graph	Description	Commit	Author	Date
	master Remove some words	c6b4376	Ben Foley <ben@...>	Today, 8:46 PM
	Add more words	2f93d45	Ben Foley <ben@...>	Today, 8:45 PM
	Add first words to wordlist	ee3346e	Ben Foley <ben@...>	Today, 8:45 PM

History

showing three commits

The **Graph** column shows the hierarchy of commits visually.

Description holds the commit message.

The **Commit** column has a unique ID for each commit. This is a "hash" of information about the commit. More on this later, for now it's enough to know that the ID is unique, and can be used to navigate through the repo history.

Author has the name and email address of the person who made the commit.

Date shows when the commit was made.

Select a commit to see which files were changed. Select a file from the bottom left panel to see the content changes in the bottom right Diff panel. Green lines prefixed by + indicate that the line was added. Red lines prefixed by - indicate the line was removed. The two columns of line numbers in the diff panel refer to the position of the change in the file before and after the change respectively.

The screenshot shows the SourceTree interface. At the top, a table lists commit history:

Graph	Description	Commit	Author	Date
	master Remove some words	c6b4376	Ben Foley <ben@...>	Today, 8:46 PM
	Add more words	2f93d45	Ben Foley <ben@c...>	Today, 8:45 PM
	Add first words to wordlist	ee3346e	Ben Foley <ben@c...>	Today, 8:45 PM

Below the table, the 'wordlist' file is selected. The diff view shows the following changes:

```

Hunk 1: Lines 1-4  Reverse hunk
1 1  beans
2 2  - milk
3 2  chocolate
4 3  salt
5 4  cinnamon

```

The commit details for the selected commit are:

Commit: c6b4376e0dc2a43e3931f6315b4ec5169af7ab2b [c6b4376]
 Parents: 2f93d451f1
 Author: Ben Foley <ben@cbmm.io>
 Date: 23 September 2017 at 8:46:05 pm AEST
 Labels: HEAD -> master

File

changes in History

10.7.4 Let's branch

Branches are a handy way to organise your changes into logical groups. In software development, branches are often used to group changes according to the features they will bring to the project. The architecture of your repository's branches is referred to as a workflow. For corpus work, branches might help organise your changes according to particular processing activities.

One of the great things about branches is that you can fiddle around with your changes without affecting other people's work in the same repo.

While it's good to have an understanding of branching, many projects are content with only one or two branches. See the [workflows](#) section above for more information about different scenarios.

We'll make our next lot of changes on a new branch.

1. In SourceTree, ensure that you have committed all of your current changes. It's best to branch from a "clean" state, where there are no uncommitted changes. Select the top-most commit and click the Branch icon. Name the branch (lowercase, no spaces, eg `process-words`) and click Create Branch.
2. Making a new branch hasn't changed our files, we have just added a new reference into our tree. We can view all our branches in the sidebar's Branches item.
3. Edit the file in VS Code (do something like sort the contents), save, add and commit it.
4. Our History now shows that the `process-words` branch is ahead of the `master` branch. Let's move it ahead even further, with a series of changes. Change the file again, add and commit after each change. Do something like capitalise the words, add and commit, then insert blank lines between each word, add and commit again. Have a look at the History now, there should be a few commits difference between the two branches.
5. Now, say we're happy with our our wordlist. Time to "merge" our changes back into `master`. Check again that there are no uncommitted changes. Change to the master branch by double-clicking the `master` branch name in the sidebar, or right-click and choose "Checkout master". The name should go bold in the sidebar. Because we are now back to the snapshot when we diverged, the contents of the file are back how they were before you did the processing work! Merging will bring the changes made on the other branch into this one.

6. Click the Merge icon. In the merge window, change the left-option list to the name of the branch you want to bring in (process-words).
7. In the options below the file pane, tick **Create a commit even if merge resolved via fast-forward**. Click OK and *Vooom*, the changes in the source branch are merged into the target. The source branch can be deleted but the change history lives on.

10.7.5 Remotes

To benefit from using Git as a collaboration tool, a copy of the repository needs to be accessible online. These remote repositories are typically hosted on github.com or gitlab.com.

Remote repositories can be cloned locally, and changes can be pushed and pulled to/from the remote. Remotes can also be “forked”, copied to another remote location.

1. First, register an account at a gitlab.com. Make sure you use the same email address that you used in the SourceTree preferences. This will authenticate you for pushing and pulling at GitLab.
 - 1.1 Verify your email if required, then sign in.
 - 1.2. Create a new project on gitlab.com. Name it `learn-git-collab`. Make it public so that SourceTree can access it. You can have private repositories, but it takes a little more effort to set up authentication for SourceTree to access them.
2. Create a working copy of a remote repository by **File > New/Clone** and choose **Clone from URL**.

Source URL: Get this from your remote repo’s HTTPS field on the overview page. Destination Path: Select the folder which contains `learn-git`.

3. Click **Clone** to download the remote repository.

The folder structure on your computer should now look like this, with the newly cloned repo next to the one we were working with earlier:



4. After cloning, SourceTree will open a window for the repository. Now the REMOTES sidebar menu has a new item “origin”.
5. The change/commit process is the same as before, so make some changes, add and commit them.
6. After committing, we see our local branch (master) advance, but the origin references (origin/master) are still on the original commit.
7. Anything you commit will update your local repository, but won’t change the remote until you push.
8. Now, try pushing your local changes to the remote. Click the Push button, and in the popup window, tick Push and OK. If you have multiple local or remote branches, you can choose what to push where from this window.
9. After the push has happened, refresh the gitlab.com window to see that the files you added have been pushed to the remote repository.

10.7.6 Protecting branches

Writing directly to a remote's master branch is convenient, but in a collaborative project you might want more control over who can write to particular branches. A common workflow is to protect the master branch from pushes and require people to make changes on their own branches. These user-created branches can be pushed back to the remote repository and used to create a merge request (aka pull request). Merge requests are then checked and the changes can be merged into master or discarded by people with appropriate access permission.

For more information about setting up a protected branch with GitLab, [see the docs](#).

To submit a merge request to a protected branch:

1. Clone the remote repo to your local computer as before.
2. Make a new branch.
3. Make your changes and commit.
4. Push just the new branch to the remote; in the Push options, choose the local branch that has the work as the source.
5. Once the push has completed, go to the GitLab project's **Repository > Branches** page.
6. Make a merge request from your work branch by clicking the **Merge request** button. From this page you can also compare the files in the commit.
7. Fill out enough details in the **New Merge Request** form to inform the reviewer what the commit does. You can assign it to a particular person (they'll get a notification when the request is made), and create labels that will help organise the request (useful if you have lots of requests open). Check that the source and target branches are as you expect. Source should be the branch that you pushed, target is usually master. To keep the repo clean, tick the option to **Remove source branch when merge request is accepted**. And you can also tick **Squash commits when merge request is accepted** to combine all the individual commit messages from the source branch into one.
8. Then click the **Submit merge request** button to create the request.
9. Now there's a merge request, anyone can comment, not just the assignee.
10. Someone with merge permissions can now view the request and merge it. If there's a conflict, the reviewer might amend the commit or send it back to the submitter with a comment to fix the changes. If there are corrections to be made, make sure to push the changes to the same branch.

10.7.7 Dealing with merge conflicts

Let's pair up for this activity.

1. Create an account on gitlab.com.
2. Person A from each pair:
 - 2.1. After signing in to gitlab.com, click the **New Project** button. Choose **Blank** as the template. Give it a name. Make it public. Click **Create project**.
 - 2.2. Add the other person as a contributor in **Settings > Members**. Search for your partner by the email address of their [gitlab](https://gitlab.com) account, set their role permission to **developer** then **Add to project**.
 - 2.3. Clone the repo and open it in VS Code.
 - 2.4. Create a new file.
 - 2.5. Commit and push in SourceTree.
3. Person B, once the first commit is showing in the remote: clone the repo.

4. Working simultaneously now... Both:
 - 4.1. Make a new branch.
 - 4.2. Edit the file. Commit it.
 - 4.3. At this point, you should have differing content! When we push and attempt to merge, the first one we do will work, the second will have a conflict.
 5. Both: push your branch to the remote and create a Merge Request from it.
 6. One person,
 - 6.1. Approve one of the merge requests.
 - 6.2. When it has merged, attempt the other. *Eeeeeek*, there are merge conflicts! When there's a conflict, a merge request can't be merged without manual intervention. Let's look at how to resolve the conflicts.
 - 6.3. Click the **Resolve conflicts** button to open a diff page [1], showing the conflict sections in the conflicting files. Conflicts are labelled as "ours" (the source) or "theirs" (the target).
 - 6.4. You can choose to keep one or the other by clicking the corresponding **Use ...** button.
 - 6.5. Or you can combine the changes manually, by clicking the **Edit inline** button.

Files with conflicts are shown with <<<<<<, ===== and >>>>>> symbols wrapping the conflicting chunks. Edit the file, removing those symbols and manually fix the text to suit what should be saved.

 - 6.6. Write a message describing the commit resolution and click **Commit conflict resolution**.
 - 6.7. Once the commit has been made, the merge request page should get a big blue banner on it. Wait a moment for the status message to update (should say something about HEAD recently changing). Reload the page at this point to make the Merge button active.
 - 6.8. Click the merge button to process the merge.
 7. Done! The branches will be removed automatically.
 8. Both: now update your local repository with the remote changes by doing a **Pull**.
- [1] The web interface may not work on large files! In which case, try resolving conflicts locally (guide tbc).
-

10.8 Definitions/glossary

AddWrites changes to a staging area.

BranchA branch is a parallel version of a repository. It is contained within the repository, but does not affect the primary or master branch allowing you to work freely without disrupting the "live" version. When you've made the changes you want to make, you can merge your branch back into the master branch to publish your changes.

CommitA commit is an individual change to a file or set of files. It's like when you save a file, except with Git, every time you commit you save information about who made the change, what was changed, and when. Each commit gets a unique ID that can be used to navigate through the repository's history.

DiffA diff is the *difference* in changes between two commits, or saved changes. The diff will visually describe what was added or removed from a file since its last commit.

FetchUpdate your local repo with status of the remote. Fetch only downloads new data from a remote repository - but it doesn't integrate any of this new data into your working files. [Read more about the difference between fetch and pull](#)

GitVersion control system for tracking changes in files.

HeadA symbolic reference to the current checked out branch.

IndexThe area where file changes are added when staging. Also known as the *staging area*.

MergeCombine the changes from one branch into another.

PullPull is similar to doing fetch-and-merge. It tries to merge remote changes with your local ones. Avoid pulling when you have uncommitted local changes.

PushSend your local changes to a remote repository.

StagingPreparing and organising a commit.

Version controlA system that saves the history of changes to files.

10.9 More definitions

- [More definitions from GitHub](#)
- [Git FAQs from GitTower](#)

10.10 Further reading

The seven rules of a great Git commit message

Comparing Git Workflows:

- [Centralized Workflow](#)
- [Feature Branch Workflow](#)
- [Gitflow Workflow](#)
- [Forking Workflow](#)

10.11 Stay tuned for more git workshops

Coming soon:

- [Setting up SSH for working with private repositories](#)
- [Differences when working with GitHub](#)
- [Working with large files](#)
- [Resetting/reverting and other git-fu](#)

THE COEDL CORPUS PLATFORM

author: Tom Honeyman date: 2017-10-05

tags: - ANNIS - corpora - transcripts

CoEDL has a “proof-of-concept” corpus platform, currently restricted to corpus contributors. This is a project aimed at making textual materials more readily available to researchers.

This is a basic guide to the platform.

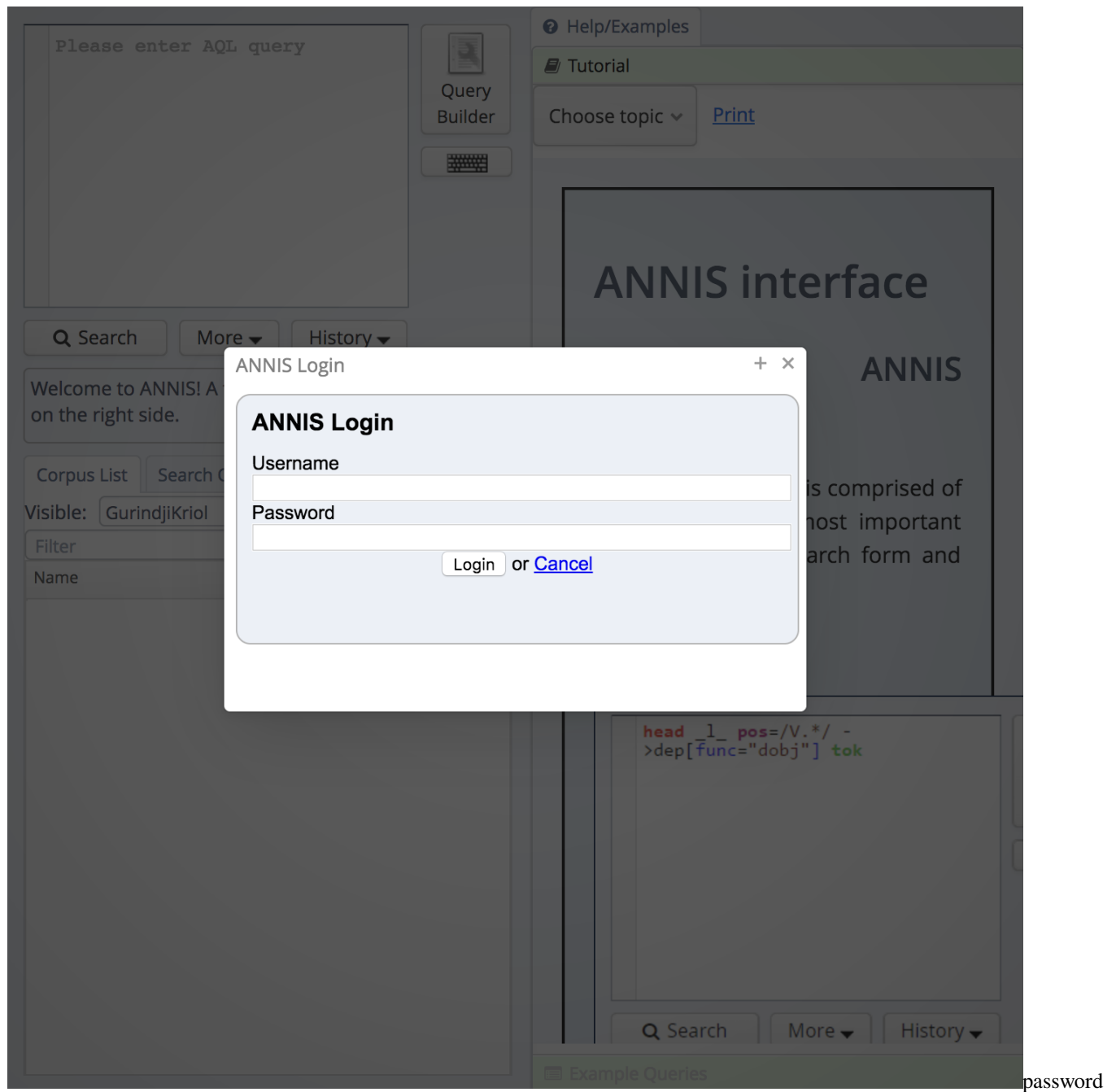
11.1 CoEDL corpus platform

The [preliminary corpus platform](#) is available, but currently limited to those with a password.

[ANNIS](#) is an open source corpus platform. A generic user guide is [available](#), while this is a simple guide to just the features available within the CoEDL version.

11.2 Entering the corpus platform

The corpus platform can be accessed at <http://go.coedl.net/corpora>. Currently access is limited to contributors, via a password. Click on the ‘Login’ button located on the top-right corner of the main page, and then enter your username and password to continue:



prompt

The main page looks like this:

The screenshot displays the ANNIS web interface. At the top, there is a navigation bar with a menu icon, 'About ANNIS', a link to 'Help us to make ANNIS better!', and a user status 'logged in as "meakins"' with a 'Logout' button. Below this, the interface is divided into several sections. On the left, there is a 'Query Builder' section with a text area for 'Please enter AQL query' and a 'Query Builder' button. Below this is a search bar with 'Search', 'More', and 'History' buttons. A welcome message states: 'Welcome to ANNIS! A tutorial is available on the right side.' Below the search bar is a 'Corpus List' section with a 'Search Options' tab. The 'Visible' dropdown is set to 'GurindjiKriol'. A table lists the corpus details:

Name	Texts	Tokens		
Gurindji-Kriol_CHAT	3	1,190	i	d

On the right, the 'Tutorial' section is active, showing the title 'ANNIS interface' and the subtitle 'Using the ANNIS Interface'. The text explains: 'The ANNIS interface is comprised of several areas, the most important of which are the search form and the results tab.' Below this, the section 'The Search Form' is shown, which includes a preview of the query builder interface with a sample AQL query: `head _l_ pos=/V.* / ->dep[func="dobj"] tok`. At the bottom of the interface, there is a green bar labeled 'Example Queries'.

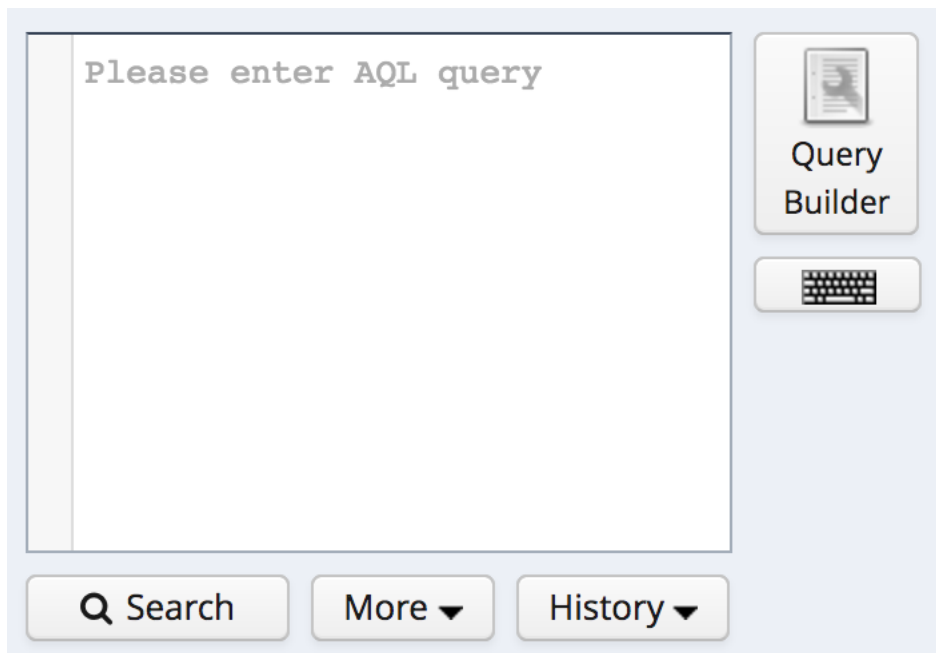
annis page

Note the logout button in the top right.

11.3 The basic layout

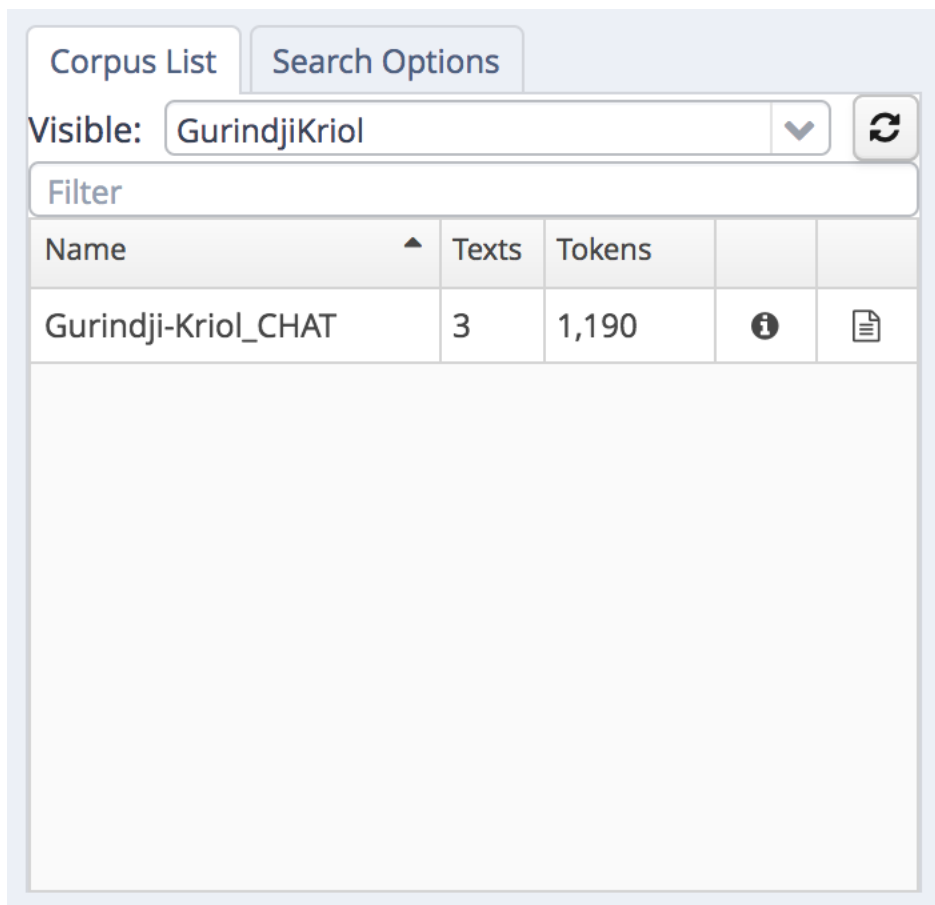
The page is made up of three components: the query panel and corpus list on the left and the results page(s) on the right.

To begin with, the query panel will be empty:



empty query panel

The (sub-)corpus panel will list one or more sub-corpora:



Name	Texts	Tokens		
Gurindji-Kriol_CHAT	3	1,190	i	

sub-corpus list

The “Visible:” drop down menu filters the sub-corpus list:

Corpus List

Search Options

Visible:

GurindjiKriol

↕

↺

Filter

All

Name

GurindjiKriol

Gurindji-Kriol_CHAT	3	1,190	<div>i</div>	<div>📄</div>
---------------------	---	-------	--------------	--------------

visible list

Corpus contributors can browse other corpora (which is encouraged, so you can see what other types of annotations others are contributing). View all available corpora by choosing “All”:

















Corpus List

Search Options

Visible:

All

Filter

Name	Texts	Tokens		
Abui-Kratchovil	26	22,335		
Anindilyakwa-Bednall	5	5,146		
BarungaKriol-Plaintext	3	36,988		
CIMaori-partialHTML	1	10,405		
CIMaori-Plaintext	1	838		
Dalabon-Ponsonnet	1	1,789		
Gurindji-Kriol_CHAT	3	1,190		
Kaytetye-Hale	1	1,382		

All sub-corpora

11.4 Searching

To search the corpora, select one or more subcorpora in the list. In the example, we are searching in the Gurindji-Kriol corpus:

select a sub-corpus

11.4.1 Searching words/tokens

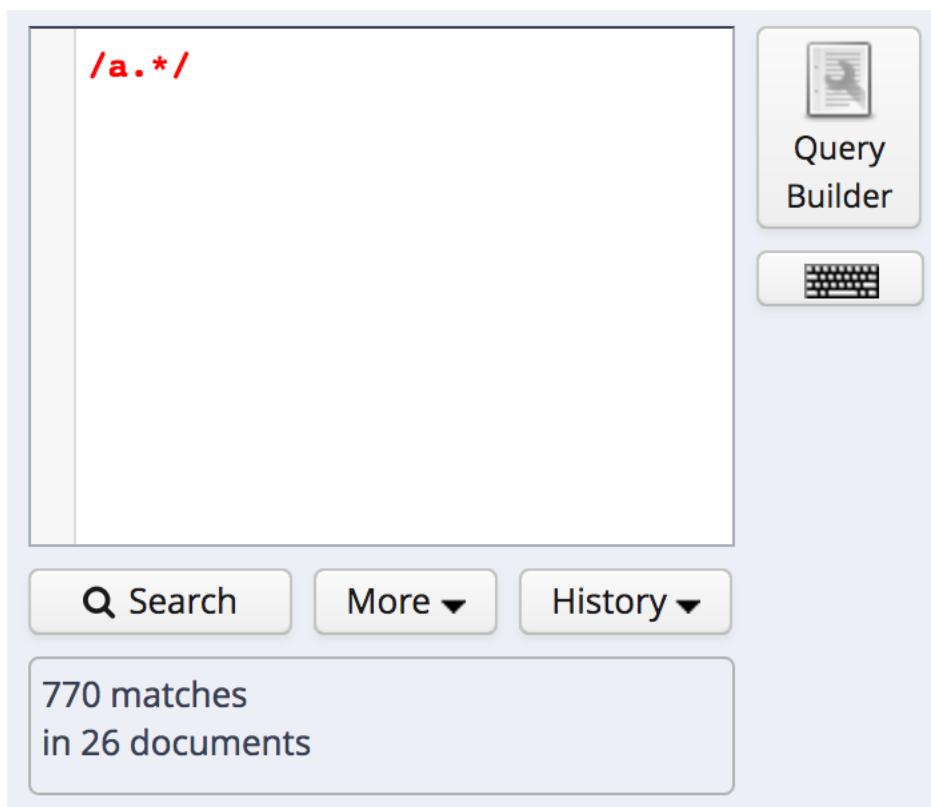
Every single corpus has a baseline layer called “tok” (for “token”). This is usually a word level representation of the primary text. It is the default layer to search on, and so a basic query can be either a search for a word (in quotes):



The screenshot shows a search interface. On the left is a large text input area containing the text **"and"** in red. To the right of this area are two buttons: "Query Builder" (with a magnifying glass icon) and a keyboard icon. Below the input area are three buttons: "Search" (with a magnifying glass icon), "More ▼", and "History ▼". At the bottom of the interface is a light blue box containing the text: "Valid query, click on "Search" to start searching."

search for a word

Or a regular expression between forward slashes (/):



/a.*/

Query
Builder

Search

More ▼

History ▼

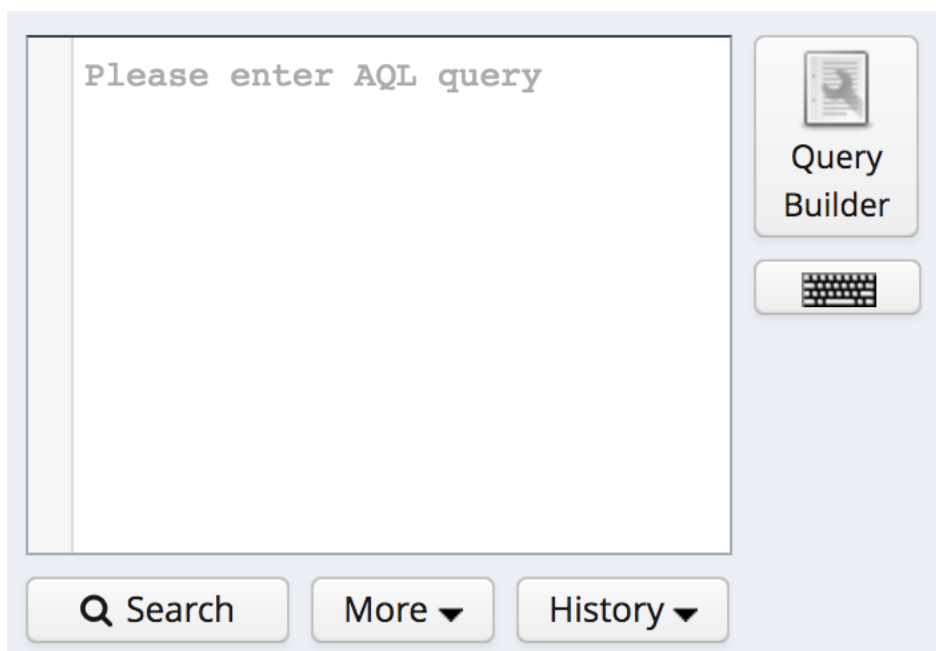
770 matches
in 26 documents

search for a-initial words

More complex searches can be built with the query builder. This is a good way to learn the full syntax of the annis query language (AQL).

11.4.2 Query builder

The easiest way to build a complex search is to use the query builder in the top left. Click on “query builder”:



Please enter AQL query

Query
Builder

Search

More ▼

History ▼

query builder

After clicking “initialise”, we can begin to construct a query.

Queries can be sequences of one or more “tokens” (i.e., annotations on a specific layer or tier). They can fall under the scope of a “span” (e.g., limited to a specific speaker). Metadata for a file can also be used to constrain the search.

In order to fall under the scope of a span, these spans must first exist in the corpus. Not all corpora have these spans. If you provided segmented text (e.g., utterances) with extra information like speaker turns or translations, then you should have annotations for these categories of information. Spans can be any grouping that interests you. For instance, spans of reported speech, of syntactic units, or of any other grouping that may be of interest to you.

Linguistic sequences

Begin by choosing the “word sequences and meta information” search, and then clicking “initialise”. Then add an element/token to a linguistic sequence. First choose which layer you want to match:

Help/Examples Query Builder ×

Word sequences and meta ▾

Linguistic sequence

Add ▾

- cm
- duration
- ft
- inf
- lemma
- mor
- pos
- speaker
- start
- tok
- tx

Toolbar

Create AQL Query Clear the Query Builder Refresh Query Builder

Advanced settings

Filtering mechanisms

generic ▾

choose

a layer

If you choose the default “tok” layer, then you type in a word/token that you’d like to match. Regular expressions can be used, but note that the regular expression must match the whole token, not part of the token.

For any other layer, a list of possible values will appear.

Linguistic sequence

The screenshot shows a user interface for building a linguistic sequence. It features a list of tokens: warlaku, wartan, was, wen, werrim, wik, windo, wukarra, and yamak. The token 'warlaku' is currently selected. Above the list is a search bar with a dropdown arrow. To the right of the list are buttons for 'X', 'Add', and '+'. Below the list, a status bar indicates '160-168/168'.

choose a token

If you want to match more than one form, add a second token to match against:

Linguistic sequence

The screenshot shows two instances of the 'Linguistic sequence' interface side-by-side. The left instance has 'warlaku' selected, and the right instance has 'smelim' selected. Both instances have a 'Regex' checkbox checked and a 'Neg. search' checkbox unchecked. Between the two instances is a dropdown menu with a period '.' and a dropdown arrow. Above the right instance are buttons for 'X', 'Add', and '+'. Below each instance is a button with a '+' and a dropdown arrow.

a second token

choose

Here we are matching on the same layer, but it is possible to match adjacency on a different layer too.

In between the two, choose the kind of adjacency relationship:

Linguistic sequence

Two panels for defining linguistic sequences. The left panel includes a 'lemma' dropdown menu currently showing 'warlaku', a 'Regex' checkbox (checked), and a 'Neg. search' checkbox (unchecked). The right panel includes a 'Regex' checkbox (checked) and a 'Neg. search' checkbox (unchecked). A tooltip is displayed over the right panel, listing the following options:

- .2 [is preceding with one token in between]
- .1,2 [is directly preceding or with one token in between]
- .* [is indirectly preceding]
- . [is directly preceding]

relationship

adjacency

Scope

If there are layers defining useful spans which tokens fall under, results can be limited to a specific scope.

First choose the relevant layer:

The 'Scope' section shows a dropdown menu with the following options:

- cm
- duration
- ft
- inf
- lemma
- mor
- pos
- speaker
- start
- tok
- tx

The text 'choose scope' is located below the dropdown menu.

And choose a value:

Scope

Change ▾

speaker ▾

☐ Regex

Close

value

scope

Metadata

Results can be limited to specific metadata values. In the same way, add a relevant metadata label, and choose relevant values:

Meta information

Add

doc

☒ F14_43_1h

☒ FM13-35_2f

☒ FM15_62_3b

X

add metadata

11.4.3 Compile the search

Once you are happy with the search criteria, click “Create AQL Query”

Toolbar

Create AQL Query

Clear the Query Builder

Refresh Query Builder

query

build

The query is translated into a search in the query box. Click search to search for values:

lemma=/warlaku/ &
lemma=/smelim/
& speaker = "FJC"
& #1 .1,2 #2
& #3_i_#1
& #3_i_#2
& meta::doc = /(F14_43_1h) |
(FM13-35_2f) | (FM15_62_3b)/

Query Builder

Search More History

full search

11.5 Search Results

Search results are shown in the panel on the right of the page:

Help/Examples Query Builder Query Result

Base text

Result for: lemma=/warlaku/ & lemma=/smelim/ & speaker = "FJC" ...

1 Path: Gurindji-Kriol_CHAT > FM13-35_2f left context: 5 right context: 5
(tokens 139 - 167)

an jei _m warlaku dat warlaku bin smel _im _bat _karra dem . warlaku bin ja

grid (default_ns)

results

Document metadata for the result can be displayed by clicking on the (i):

Info for salt:/Gurindji-Kriol_CHAT/FM13-35_2f

Metadata

document: FM13-35_2f

Name	Value
annis:doc	FM13-35_2f

metadata

document

By default, the baseline text is shown. By clicking on “grid”, a layered display of the corpus is shown instead:

_in _at _karra na . an dat bee-hive im bi jarrei an jei _m warlaku dat warlaku bin smel _im _bat _karra dem . warlaku t

grid (default_ns)

duration	3.065					6.78		
ft	They were calling out then .					And the bee hive was that way and the do		
inf	CONT	TEL						
lemma	_in	_aut	_karra	na		en	dat	beel
mor	suf:cont _in&CONT&k=CONT	suf:tel _aut&TEL&k=out	suf:cont _karra&g=CONT	interj na&k=now	.	conj en&k=and	dem dat&k=the	n:inc
pos	suf:cont	suf:tel	suf:cont	interj	null	conj	dem	n:inc
speaker	FJC					FJC		
start	82.4					86.951		
tx	dei bin jing _in _at _karra na .					an dat bee-hive im bi jarrei an jei _m warlaku		
tok	_in	_at	_karra	na	.	an	dat	bee-

grid

view

11.6 Document browser

A basic document browser is implemented for each of the corpora. At present the display of the text is fairly basic, but will be improved upon soon.

View the document listing for a given corpus by clicking on the documents/files icon next to the sub-corpus name.

Help/Examples	Query Result	Gurindji-Kriol_CHAT	
Filter documents by name			
document name	corpus path	visualizer	info
F14_43_1h	Gurindji-Kriol_CHAT > F14_43_1h	full text	
FM13-35_2f	Gurindji-Kriol_CHAT > FM13-35_2f	full text	
FM15_62_3b	Gurindji-Kriol_CHAT > FM15_62_3b	full text	

document

listing

Click on “full text” to view each individual text.

Help/Examples	Query Result	Gurindji-Kriol_CHAT	Gurindji-Kriol_...
Gurindji-Kriol_CHAT > F14_43_1h - Visualizer: full text			
<p>dat warlaku _ngku i i bin luk dat dat dat . boi _wan _tu i bin luk na +" &yu &&are &&sleeping . dat warlaku _ngku i bin i bin gon na . xxx . dat frog _tu i bin gon na ran _awei i bin gon wen jei bin jilip _in na . an dat _tu dog an dat man dat boi _wan i bin luk na an i bin gon na . +" ai bin wer _ing mai &shoes an ai bin gon na faind _im ebri _weya na . i bin gon na . i bin gon da windou na xxx . i bin jing _in _at _karra na . i bin jing _in _at an dat warlaku bin ting krai _ing . an i bin luk na i bin luk loda bi na . an im luk an im jing _in _at deya la hol _ta na an im gu la tri na dat warlaku _ngku luk _aran bo frog deya la tri . dat mawujimawuji _ngku kam na dat warlaku _ngku bin sheik _im _bat _karra . dat warlaku _ngku i bin breik _im dat tri _na dat ting bi _ngku houm . i bin sheik _im _bat, dat wan boi _ngku bin gon insaid dat tri _ngka . luk _aran bo canetoad i neba bin luk im . im luk luk luk na i bin luk na . i bin ran na an ran _ing dat dat warlaku _ngku na i bin &scared . i bin luk sam _bodi deya na teik _im im, dat warlaku bin jing _in _at _karra . i bin juk _im im dat warlaku an dat man . an abta na i bin luk i bin juk _im dat tubala . an an abta na dei bin juk _im na la riba _ngka . an dat warlaku i was bi la hed . hed _ta bo dat man an dat man _tu was xxx ting jidan _bat . an dat warlaku _ngku i bin top kwait i bin gon na dat tri _ngka im gu la tri _ngka na . an abta na i bin luk na najing . an abta, dat warlaku _ngku na an dat man _tu i bin faind _im dat frog na . i was tumeni brog deya .</p>			

example

document

QUICK CLAN MORCODE LOOKUP

author: Sasha Wilmoth date: 2017-10-12

tags: - Tutorial - CLAN - CHAT - Gurindji Kriol - Python

categories: - Scripts

12.1 Introduction

Each utterance in the Gurindji Kriol corpus has a tier with morphological information for each token in the transcription tier. It looks like this (with the \P marking a pronominal subject):

```
*FSO:      kayirra _k tubala karrap .
%mor:      adv:loc|kayirra&g=north case:all|_k&g=ALL
           pro|dubala&3DU&k=those_two\P v:tran|karrap&g=look_at .
%eng:      Those two are looking to the north.
```

Whereas the lexicon that all these codes are taken from looks like this:

```
...
_k {[scat case:all]} "_k&g" =ALL=
_k {[scat der:fact]} "_k&g" =FACT=
...
karrap {[scat v:tran]} "karrap&g" =look_at=
...
kayirra {[scat adv:loc]} "kayirra&g" =north=
...
tubala {[scat pro]} "dubala&3DU&k" =those_two=
...
```

As you can imagine, if you have to make any small corrections to the mor tier, it's incredibly fiddly and time-consuming to look up each morph in the lexicon and type out the code. The only other option is to run the MOR command again, which is even more undesirable.

So, I wrote a little interactive script (printMorCodes.py) that looks them up for you.

12.2 Instructions

12.3 Requirements

This script requires Python 2.x. It works on Mac and has not been tested on Windows

The script can be found [here](#).

12.4 Running the script

The command is:

```
morCodeLookup.py -l lexicon(s)
```

Gurindji Kriol uses two lexicon files, so the command I use is:

```
morCodeLookup.py -l /Users/swilmoth/Dropbox/appencoedlinternship/kri/lex/lex_gurindji.  
↪cut /Users/swilmoth/Dropbox/appencoedlinternship/kri/lex/lex_kriolgen.cut
```

The script has a little welcome message, and then you just type a sentence into the terminal and it looks up the codes for you.

If you type *kayirra _k tubala karrap*, it gives you:

```
adv:loc|kayirra&g=north case:all|_k&g=ALL^der:fact|_k&g=FACT pro|dubala&3DU&k=those_two  
v:tran|karrap&g=look_at.
```

If you type *jei _m gon Lajamanu _ngkirri!* ‘They went to Lajamanu!’, you get:

```
pro|dei&3PL/S&k=they suf|_m&TAM&k=PRS v:intran|gu&k=go^v:minor|gon&k=go n:prop|Lajamanu  
case:all|_jirri&g=ALL !
```

I’ve tried to replicate CLAN’s MOR command, so punctuation is handled in a similar way, homographs have all the options listed with a ^ and anything starting with a capital letter has n:prop. And if you type something that’s not in the lexicon, you get something like:

```
Not-in-lexicon:supercalifragilisticexpialidocious
```

When you’re done, simply type `exit`.

12.4.1 Copying to clipboard

To save myself the step of highlighting the mor-codes and pressing command+C, I added an option so that when you type in *kayirra*, it not only prints `adv:loc|kayirra&g=north` to your terminal, it also copies the code to your clipboard. So you can quickly jump back to your transcript and paste it in. When you’re entering the command for the script, just add `-c`.

12.4.2 Setting up an alias

If this is something you want to use all the time, you can add an alias to your `bash` profile so you don't have to type the whole command and find the lexicon files each time. I open up my `~/ .bashrc` file, and add this line:

```
alias lookup = 'morCodeLookup.py -l /Users/swilmoth/Dropbox/appencoedlinternship/kri/lex/  
↪lex_gurindji.cut /Users/swilmoth/Dropbox/appencoedlinternship/kri/lex/lex_kriolgen.cut'
```

Next time, the only command I need is `lookup`, or `lookup -c`.

VARIOUS SCRIPTS FOR CLEANING UP THE MOR TIER (CLAN)

author: Sasha Wilmoth date: 2017-11-17

tags: - CLAN - CHAT - Gurindji Kriol - Python - Appen

categories: - Scripts

13.1 Introduction

In this post, I'll introduce 8 (yes, eight) scripts for cleaning up the %mor tier in a CLAN corpus. Fair warning: this post is pretty technical and very long. But if you want a squeaky clean CLAN corpus, you've come to the right place.

All these scripts have all been developed for the specific needs of the Gurindji Kriol corpus, and as such might need to be tweaked slightly to suit your needs. If you think these might be helpful for you but you need some help running them or making minor changes, I'm more than happy to help.

You can download all scripts [here](#), except for splitFiles.py and morChecker.py which are [here](#). All are written in Python 2.7, for use on a Mac. With the exception of morChecker.py which can be run at any point, I highly recommend following the order of scripts given in this post.

[Script|Description|Input|Output| — | — | — | — | morChecker.py | Finds anomalous mor-codes with comparable contexts (of a user-determined length) in a corpus|Corpus location|Table (copy into a spreadsheet) | validateMorCodes.py | Checks that the mor-codes in a corpus all look okay according to your lexicon(s)|Lexicon(s), corpus location|2-column file for correction | correctMorCodes.py | Fixes any known mor-code errors, based on the output of the above.|Corpus location OR giant text file with entire corpus, 2-column file with corrections|New versions of any corrected .cha files | countTokens.py | Reports instances where there's a different number of tokens on the transcription tier and on the mor tier.|Giant text file with entire corpus|Table (copy into a spreadsheet) | checkMorcodeMatches.py|Checks that the mor-codes actually correspond to the transcription.|Lexicon(s), giant text file with entire corpus|3-column file for correction | correctMismatches.py|Fixes known mis-matches, based on the output of the above.|Giant text file with entire corpus, 3-column file with corrections|Giant text file with entire corpus (corrected) | insertPlaceholders.py|Adds missing tokens to the mor tier that aren't added by CLAN's MOR command ('xxx', foreign words, etc)|Giant text file with entire corpus, 3-column file with corrections|Giant text file with entire corpus (corrected) | splitFiles.py|Splits up that giant text file and puts the files back into place|Giant text file with entire corpus|Individual files!

13.2 About that giant text file...

Some things are easier to fix when you just have one file open, instead of trying to manage hundreds in various locations. At different stages of this process, I've been joining all the .cha files into one text file, and splitting them back up again. I join them all together using grep:

```
grep ' ' */*.cha */*.cha > ALLFILES
```

Running this command from the top directory of my corpus joins together all the .cha files (in 1 or 2 nested subdirectories). It keeps the relative path of each file at the start of each line, like this:

```
../../../../GurindjiKriol/Old/FHM002/FHM002.cha:
../../../../GurindjiKriol/Old/FHM002/FHM002.cha:*FAC: an i bin gib _it to dat dadi _wan na . ^U397203_400059^U
../../../../GurindjiKriol/Old/FHM002/FHM002.cha:%mor: con|en&k=and pro|i&3SG/S&k=he/she/it\P v:aux|bin&PAST&k=PST
../../../../GurindjiKriol/Old/FHM002/FHM002.cha: v:tran|gib&k=give suf|_im&TR&k=TR prep|to&k=ALL
../../../../GurindjiKriol/Old/FHM002/FHM002.cha: dem|dat&k=the n:kin|dedi&k=father der:nom|_wan&NOM&k=NMLZ
../../../../GurindjiKriol/Old/FHM002/FHM002.cha: interj|na&k=now .
../../../../GurindjiKriol/Old/FHM002/FHM002.cha:*FAC: an den i bin gib _it im dat Sprite . ^U403967_406661^U
../../../../GurindjiKriol/Old/FHM002/FHM002.cha:%mor: con|en&k=and con|den&k=then pro|i&3SG/S&k=he/she/it\P
../../../../GurindjiKriol/Old/FHM002/FHM002.cha: v:aux|bin&PAST&k=PST v:tran|gib&k=give suf|_im&TR&k=TR
../../../../GurindjiKriol/Old/FHM002/FHM002.cha: pro|im&3SG&k=he/she/it dem|dat&k=the n:prop|Sprite .
../../../../GurindjiKriol/Old/FHM002/FHM002.cha:@End
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha:@UTF8
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha:@Color words: * 1 36683 46021 4742
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha:@Begin
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha:@Languages: gk
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha:@Participants: FAC Azaria Speaker, FJC Jasmine Speaker, FFM
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha: Felicity Investigator
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha:@ID: gk|meakins|FAC|12;00.00|||Speaker|||
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha:@ID: gk|meakins|FJC|09;00.00|||Speaker|||
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha:@Media: gk|meakins|FFM| |||Investigator|||
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha:@Location: FHM002, audio
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha: game 10:11min. Azaria playing with Jasmine using the locative b
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha:@Date: 16-SEP-2004
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha: Jasmine yu garram dat wan kajirri im gon ontop s
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha:%mor: n:prop|Jasmine pro|yu&2SG&k=you\P
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha: v:tran|garram&k=have
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha: dem|dat&k=the dem|wan&k=a
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha: n:human|kajirri&g=older_woman pro|im&3SG&k=he/she/it\P
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha: v:intran|gu&k=go adv:loc|ontop&k=above
../../../../GurindjiKriol/Old/FHM002/FHM002_named.cha: n:place|shop&k=shop case:loc|_ta&g=L0C .
```

Giant

text file

I know it looks a bit crazy, but I promise works! See [splitFiles.py](#) for details on how to split this back up again.

Note: the file shown in the screenshot would have used a different command, like `grep ' ' ../../GurindjiKriol/Old/*/*.cha`

13.3 morChecker.py

13.3.1 Description

Disambiguating homographs on the mor tier is a pain. You can easily introduce heaps of errors, but it's almost impossible to find them. This script (by Simon Hammond), checks the consistency of mor-coding in similar contexts, to find errors such as a verb being coded as intransitive when it is followed by a transitive suffix (to mention a more straightforward example). This script looks at the mor-tier, and finds sequences of mor-codes of any given length, in which the forms are identical, but other information in the mor-codes differs. I run this script over all new data with different chosen lengths, and sort the output by frequency to find likely errors. The following table shows a small, selected excerpt of the output when searching for 4-grams:

[Total	freq]	Phrase]	Lemma]	POS	1]	Freq	1]	POS	2]	Freq	2]	POS	3]	Freq	3]	— — — — — — — —					
—	120]	START	en	dat	ting ting n	118]	interj	2]	74]	i	bin	jak	_im jak v:tran	68]	v:intran	6]	20]	dei	bin	gu	ged

_im|gu|v:intran|1|v:minor|19|| |19|dei bin gu hant|gu|v:intran|1|v:minor|18|| |35|dei bin ting na|ting|n|1|v|34|| |19|i bin ran _wei|_wei|der:nom|2|suf:tel|17|| |15|bin meik _im _bat|meik|v:tran|1|v:intran|2|v:minor|2|

This table tells us the total frequency of the particular sequence in question. The beginning and end of an utterance are also treated as parts of the sequence, as this may be salient information. It also tells us which item has been found with inconsistent mor-coding. Then, it tells us the POS information of each different mor-code found for that form, and how often each mor-code occurs. There are also extra columns that show the full mor-code sequence, so we can spot other inconsistencies (such as when the English gloss differs but the POS is the same). Going through this list:

- *ting* in *dat ting* should always be coded as ‘n’ not ‘interj’, due to the article *dat*.
- *jak* in *jak _im* should always be coded as ‘v:tran’, because of the transitive suffix *_im*
- *gu* in *gu ged _im* should always be ‘v:minor’.
- Likewise for *gu hant*
- *ting* in *bin ting* should always be ‘v’, not ‘n’, due to the past tense *bin*
- *_wei* in *ran _wei* should always be ‘suf:tel’
- *meik* in *meik _im _bat* should never be ‘v:intran’. It could be either ‘v:tran’ or ‘v:minor’ depending on the context - we might like to check this.

Based on this, we can make a list of automated fixes to make across the whole corpus, or search for particular examples that seem strange.

This process will not catch every coding error, but it goes a long way and gives a good overview of the data. For best results, I run the script several times with shorter n-gram lengths each time. Searching for shorter n-grams results in an overwhelming output, in which the majority of inconsistencies are legitimate differences. This is still the case for most of the 4-grams identified, but it’s possible to look through them and find likely errors. Searching for longer n-grams gives a better signal-to-noise ratio in the output (i.e. inconsistencies found when searching for 10-grams are much more likely to be errors), but a much smaller output.

13.3.2 Limitations

One limitation of this script is that it only looks at the citation form in the mor-code, and not the actual form that has been transcribed. Therefore, it won’t detect errors where *garra* was inconsistently coded as *pre|garra&ASS&k=PROP* when it should have been *v:aux|garra&k=have*, as the two citation forms are different. It does have one exception built in: the ergative and dative suffixes are treated as identical. They have two homophonous allomorphs (*_ngku* and *_tu*), but their citation forms are *_tu* and *_ku* respectively, so would not be found by the script. This exception was included because the changing case system is of particular interest and they are easily mis-coded. A future version of this script could take into account the transcription tier, and would be run after all these other cleaning up scripts are run.

13.3.3 Instructions

The arguments of this script are: the location of the .cha files, and the desired length of the n-gram (if no length is given, the default is 4). The script compares any files with a .cha extension in all subdirectories.

```
morChecker.py /path/to/your/corpus/ -n [a number of your choice]
```

There’s also an option for a simpler output (add *-s* to your command), which just prints POS information and not the full codes.

13.4 validateMorCodes.py

This script is pretty simple - it checks that all the mor codes in your corpus are valid according to your lexicon(s). It ignores things that aren't in the lexicon, like punctuation and proper nouns. The script outputs a two-column file - you can correct the second column and automatically fix the corpus with correctMorCodes.py (below). If you've used this script before and already have a list of known incorrect mor-codes, use the `-c` option to ignore them. The lexicons should be in the `.cut` format, which looks like this:

```
@UTF8
_abat {[scat suf:tel]} "_abat&k" =about=
_abta {[scat suf:tel]} "_abta&k" =after=
_alang {[scat suf:tel]} "_alang&k" =along=
_am {[scat suf]} "_im&k" =TR=
_an {[scat suf:tel]} "_an&k" =on=
_ap {[scat suf:tel]} "_ap&k" =up=
...
```

The options of the script are:

```
-h  help message
-l  any number of .cut lexicons
-c  checked codes that you want to ignore (optional)
-d  directory containing .cha files (it will search in all subdirectories)
```

So my command is:

```
validateMorCodes.py -l lex_gurindji.cut lex_kriol.cut -c checkedMorCodes.txt -d /path/to/
→my/corpus
```

The output is a two-column (tab-delimited) file. Here's a snippet of mine - I've corrected the second column according to the current lexicon:

? Wave-Hill	n:prop Wave-Hill
? and	conj en&k=and
adv:loc karlarra&g=from_west	adv:loc karlarra&g=west
n:bp blud&g=blood	n:bp blud&k=blood
num jarrwa&g=big_mob	n jarrwa&g=big_mob
num jintaku&g=one	n jintaku&g=one
v:intran yamak&g=slow	adv yamak&g=slow
v:tran basim&k=pass	v:tran pajim&k=pass
v:tran nok&k=hit_head	v:tran nok&k=hit
...	

This file then becomes the input for...

The script ignores certain stuff that we expect to be missing - e.g. *xxx* for unintelligible speech, things like *&=laugh* or *&foreign &words*, etc. If you've already run `insertPlaceholders.py` (see below) and want to run `countTokens.py` again, add a `-p` to the command.

13.7 checkMorCodeMatches.py

After running the previous scripts, all the mor-codes match existing entries in the lexicon, and the transcription and mor tiers have the same amount of stuff on them. At this stage, we can check that every single token on the transcription tier actually matches up with its corresponding code. **Don't run this unless the output of `countTokens.py` is 0!**

The inputs for this script are the `.cut` lexicon(s) and that giant text file. This script also ignores tokens like *xxx*, *&=laugh*, etc. The command is:

```
checkMorcodeMatches.py -l MyLexicon.cut [optional_other_lexicons.cut] -i MyEntireCorpus.
↪txt > mismatches.txt
```

The output is a 3-column tab-delimited file for correction. The first column is the token on the transcription tier. The second column is the corresponding mor-code that doesn't match the lexicon. The third column is a suggestion based on the lexicon. Go through this file carefully and correct the third column.

im	pro i&3SG/S&k=he/she/it	pro im&3SG&k=he/she/it
liar	pro yu&2SG&k=you	n:human laiya&k=liar
partiki	case:abl _partak&g=ABL	n:inanimate partiki&g=nut_tree
poniteil	adj parlarra&g=bald	n:bp poniteil&k=ponytail
mami	n:prop Mami	n:kin mami&k=M
gu	v:minor gu&k=go	v:minor gon&k=go
birdi	n:animal bird&k=bird	n:animal birdi&k=bird
yu	pro i&3SG/S&k=he/she/it	pro yu&2SG&k=you
_ta	adv:loc deya&k=there	case:loc _ta&g=LOC

checkMorCodeMa

In the screenshot above, you can see some really minor differences (e.g. the lemma of *gu* should be *gon* when it's a minor verb), and some instances where the transcription was corrected but not the mor-code (someone must have heard *_partak* at first, but changed that to *partiki* later). And some things that are just wrong, like *liar* has the mor-code for *yu*. This is actually a good way to find odd inconsistencies in the lexicon - fix any you find and run the script again. Once we're happy with the third column, we can correct these automatically with...

13.8 correctMismatches.py

This script corrects those mismatches according to our 3-column file. The input must be that giant text file. The command is simple:

```
correctMismatches.py -i MyEntireCorpus.txt -m mismatches.txt > MyEntireCorpus_Corrected.
↪txt
```

The script will **only** make corrections when that particular mismatch occurs. That is, `adv:loc|deya&k=there` will only be changed to `case:loc|_ta&g=LOC` when `_ta` is the corresponding token on the transcription tier. It also ignores tokens like *xxx* and *&=laugh*.

Because this script relies on the connection between the two tiers, make sure you have completely cleaned up any errors found with `countTokens.py`.

13.9 insertPlaceholders.py

So far, these scripts have been ignoring things that don't get added to the mor tier by CLAN, like *xxx*, *&=laugh*, *&foreign &words*, and *[uncertain words]*. But if we're converting to another format where the connection between each token on each tier needs to be explicit, or if we want to include them for the sake of accurate searching and analysis, then we need to include them on the mor tier.

It turns this:

```
*FFM: LD &yu &gonna &eat _im &all &_ap &hey .
%mor: n:prop|LD suf|_im&TR&k=TR .
```

into this:

```
*FFM: LD &yu &gonna &eat _im &all &_ap &hey .
%mor: n:prop|LD &yu &gonna &eat suf|_im&TR&k=TR &all &_ap &hey .
```

This is just an example to show how the script works - please excuse the mixed up English/Kriol spelling!

This script is super simple to run:

```
insertPlaceholders.py MyEntireCorpus.txt > MyEntireCorpus_placeholders.txt
```

To ensure that the script inserts these placeholder tokens in the right spot, **only** run this script after you have run `countTokens.py` and `checkMorCodeMatches.py`, and the output of both is 0.

13.10 splitFiles.py

After running the above scripts, we can be confident that every single mor code in the entire corpus is up to date with the lexicon, matches every single morph that has been transcribed, and is as clean and consistent as we can possibly make it. We just need to split up that giant text file and put each file back into place.

This script was written by Simon Hammond and Stephanie Soh at Appen - CoEDL-affiliated people, you will need to request permission to use it.

This script has two options: you can just output all the files in one folder, or you can put them back in their original places. The latter will almost certainly overwrite whatever version is already there, so proceed with extreme caution.

To output all files into a single folder:

```
splitFiles.py MyEntireCorpus.txt -o /path/to/cleaned/up/corpus/
```

The `-o` part is optional - if you leave it out, the script defaults to the current directory.

To put them back in their original locations:

```
splitFiles.py MyEntireCorpus.txt --replace -o /path/to/relative/location/
```

You can use the `-o` option in combination with the `--replace` option if the paths in your giant text file are *relative* paths. Here's a screenshot of what that text file looks like again:

```

.././GurindjiKriol/Old/FHM002/FHM002.cha:
.././GurindjiKriol/Old/FHM002/FHM002.cha:*FAC: an i bin gib _it to dat dadi_wan na . ^U397203_400059^U
.././GurindjiKriol/Old/FHM002/FHM002.cha:%mor: conj|en&k=and pro|i&3SG/S&k=he/she/it\P v:aux|bin&PAST&k=PST
.././GurindjiKriol/Old/FHM002/FHM002.cha: v:tran|gib&k=give suf|_im&TR&k=TR prep|to&k=ALL
.././GurindjiKriol/Old/FHM002/FHM002.cha: dem|dat&k=the n:kin|dedi&k=father der:nom|_wan&NOM&k=NMLZ
.././GurindjiKriol/Old/FHM002/FHM002.cha: interj|na&k=now .
.././GurindjiKriol/Old/FHM002/FHM002.cha:*FAC: an den i bin gib _it im dat Sprite . ^U403967_406661^U
.././GurindjiKriol/Old/FHM002/FHM002.cha:%mor: conj|en&k=and conj|den&k=then pro|i&3SG/S&k=he/she/it\P
.././GurindjiKriol/Old/FHM002/FHM002.cha: v:aux|bin&PAST&k=PST v:tran|gib&k=give suf|_im&TR&k=TR
.././GurindjiKriol/Old/FHM002/FHM002.cha: pro|im&3SG&k=he/she/it dem|dat&k=the n:prop|Sprite .
.././GurindjiKriol/Old/FHM002/FHM002.cha:@End
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:@UTF8
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:@Color words: * 1 36683 46021 4742
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:@Begin
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:@Languages: gk
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:@Participants: FAC Azaria Speaker, FJC Jasmine Speaker, FFM
.././GurindjiKriol/Old/FHM002/FHM002_named.cha: Felicity Investigator
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:@ID: gk|meakins|FAC|12;00.00|||Speaker|||
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:@ID: gk|meakins|FJC|09;00.00|||Speaker|||
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:@ID: gk|meakins|FFM|||Investigator|||
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:@Media: FHM002, audio
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:@Location: Azaria playing with Jasmine using the locative b
.././GurindjiKriol/Old/FHM002/FHM002_named.cha: game 10:11min.
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:@Date: 16-SEP-2004
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:*FAC: Jasmine yu garram dat wan kajirri im gon ontop s
.././GurindjiKriol/Old/FHM002/FHM002_named.cha:%mor: n:prop|Jasmine pro|yu&2SG&k=you\P
.././GurindjiKriol/Old/FHM002/FHM002_named.cha: v:tran|garram&k=have
.././GurindjiKriol/Old/FHM002/FHM002_named.cha: dem|dat&k=the dem|wan&k=a
.././GurindjiKriol/Old/FHM002/FHM002_named.cha: n:human|kajirri&g=older_woman pro|im&3SG&k=he/she/it\P
.././GurindjiKriol/Old/FHM002/FHM002_named.cha: v:intran|gu&k=go adv:loc|ontop&k=above
.././GurindjiKriol/Old/FHM002/FHM002_named.cha: n:place|shop&k=shop case:loc|_ta&g=L0C .

```

Giant

text file

As you can see, the `.././` at the start indicates we ran the `grep` command from some other subdirectory, so I would either navigate to that directory, or refer to it with the `-o` option.

13.11 Acknowledgements

All Gurindji Kriol data shown above has been collected by Felicity Meakins and Cassandra Algy. All scripts were written by me (Sasha Wilmoth), with the exception of `morChecker.py` (Simon Hammond) and `splitFiles.py` (Simon Hammond and Stephanie Soh). Many thanks for the support of Appen, in particular Simon Hammond and Jason Johnston for their help with Python.

SET UP NECTAR

author: Nay San date: 2017-12-11

tags: - Nectar - SSH - Setup

categories: - Tutorials

14.1 Nectar setup

14.1.1 About

The primary purpose of this tutorial is to be referred from other tutorials making use of the Nectar system. Here I give a quick introduction to:

1. logging into Nectar Dashboard for the first time
2. launching an instance on Nectar
3. logging into the *launched instance*

14.1.2 0. Have your SSH key ready

If you do not have an SSH key already generated for your computer, follow Step 1 of this guide: <https://kb.dynamicsoflanguage.edu.au/contribute/setup/>. The end of step 1 requires you to copy a generated public key ready for pasting elsewhere.

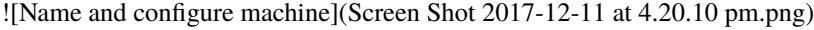
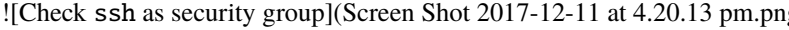
14.1.3 1. Log into Nectar and add SSH key

You can login and access the Nectar Dashboard by going to <https://dashboard.rc.nectar.org.au/>, and authorising yourself through your institution's credentials.

1. After logging in, go to the **Access and Security** page (left menu), and click **Import Key Pair** (top right):
![Import Key Pair](Screen Shot 2017-12-11 at 3.53.01 pm.png)
2. Paste your copied **Public Key** from Step 0 into the text area for the public key, and give the key a meaningful name which identifies the computer you generated the key on (i.e. `ns-mbp-2016` is for my 2016 Macbook Pro).
![Name pasted key](Screen Shot 2017-12-11 at 4.12.08 pm.png)

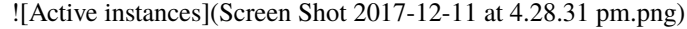
You will only have to perform this step for every new computer from which you wish to access Nectar instances.

14.1.4 2. Launch Nectar instance, with SSH Security Group checked

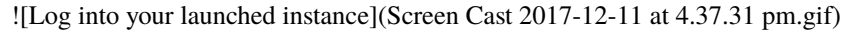
1. Go to the **Instances** page (left menu), and click **Launch Instance** (top right).
2. Name and configure the instance. For this demo, just select one of the official Ubuntu images from Nectar (e.g. NeCTAR Ubuntu 16.04 LTS (Xenial) amd64):
 (Screen Shot 2017-12-11 at 4.20.10 pm.png)
3. Make sure you check **ssh** in the **Access & Security** tab of the Launch Instance window:
 (Screen Shot 2017-12-11 at 4.20.13 pm.png) **Note.** if you want to serve out web data from the instance, you should also check **http** (port 80).

14.1.5 3. Log into your launched instance, e.g. nectar-demo

After Step 2, you should see an active instance with an assigned IP address in your instances page:

 (Screen Shot 2017-12-11 at 4.28.31 pm.png)

In a Terminal window, type `ssh ubuntu@144.6.226.20`, where `144.6.226.20` is the IP address Nectar assigned to the launched instance (you will be asked whether you really want to connect when connecting for the first time, answer yes).

 (Screen Cast 2017-12-11 at 4.37.31 pm.gif)

Once you've logged in, you'll notice that the prompt is of the form `ubuntu@nectar-demo` (i.e. `ubuntu` + `@` + name of instance), and not that of your local user/computer. Now you're ready to do things within the instance!