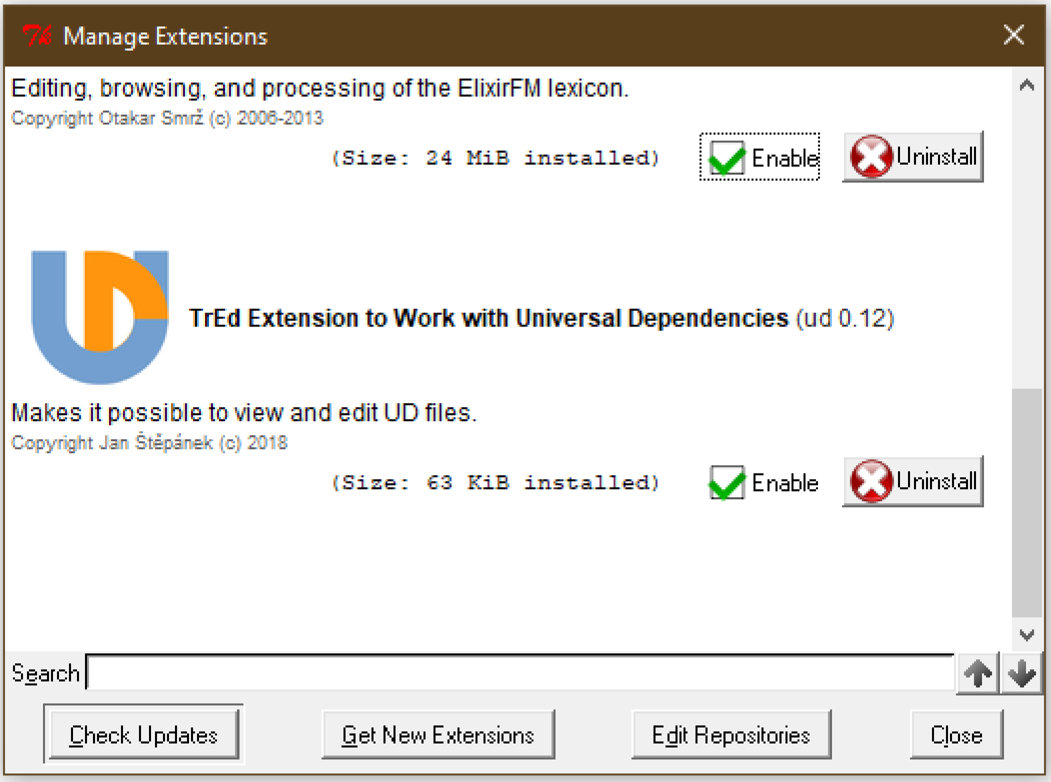
Working with UD data (practical session for NPFL075)

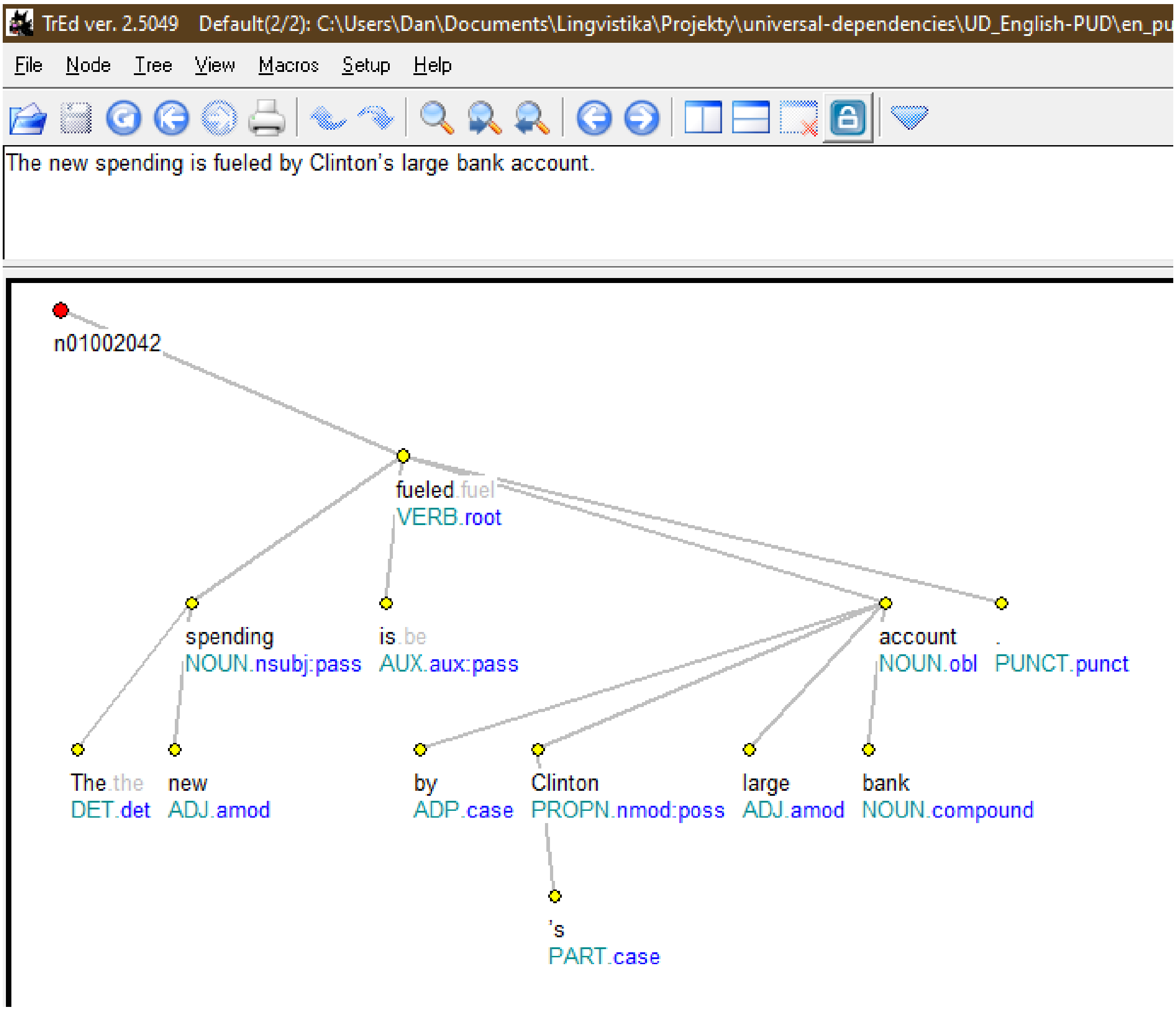
Daniel Zeman (zeman@ufal.mff.cuni.cz)

Universal Dependencies (UD) annotated data is stored in a format called CoNLL-U (https://universaldependencies.org/format.html). It is a vertical, table-like format (i.e., not XML), which is relatively easily readable even for humans.

There are various freely available tools that support this file format (https://universaldependencies.org/tools.html). You can use Tred to visualize UD data if you install the corresponding Tred extension.



Once the extension is installed, the “*.conllu” file type will appear in the File / Open dialog box. You can obtain UD data in two ways. Either you download the entire UD release from the Lindat repository (go to https://universaldependencies.org/, follow the “Download UD treebanks” links). The TGZ package is worth several hundred megabytes. If you do not want to use so much bandwidth and disk space, you can download individually only the languages you are interested in. Back on the UD title page, click the language line, then select one of the treebanks (if there are multiple UD treebanks for the language), then find the link “Repository **master**”. You will land in a Github repository whose master branch contents is identical to the latest official UD release of that treebank. You can clone that repository via Git or you can download its contents as a zipped package. Unpack it and open one of the CoNLL-U files in Tred:



Udapi

We will work with the UD data using a tool/framework called Udapi (https://udapi.github.io/). It takes care of reading and writing the CoNLL-U file, and it provides your code with an API, i.e., methods of accessing the tree structure and attributes of individual nodes. The actual code that searches the trees and/or modifies them is organized in *blocks*. Any number of blocks can be applied in sequence to the same tree, gradually modifying the data.

Udapi is available in three programming languages: Python, Perl, and Java. However, most people use the Python version and the other two do not seem to be actively developed. Also, a number of useful blocks is available for the Python version but not for the other two (note that the block would have to be converted to the syntax of the other programming language if it shall be used with Perl or Java).

The main developer of Udapi, Martin Popel, has created a tutorial at https://udapi.github.io/tutorial/. It will help you install the Python version (once installed, the Python command to use is spelled "udapy"). Note that the sample data referred in the tutorial step 2 is taken from UD release 2.0, which is now outdated (and there is another version of the tutorial on the web, with even more outdated sample data). I recommend using the latest release instead (see the previous section on how to obtain such data). Assuming that you have downloaded the UD_English-PUD treebank and you are in Bash in the folder of that treebank, you can use Udapi to visualize the tree using fixed-font characters and terminal colors:

```
cat en_pud-ud-test.conllu | udapy -T | less -R
```

```
docname = n01001
loaded_from = -
# sent_id = n01001011
# text = “While much of the digital transition
ote in a blog post Monday.
─┐
 │ ┌── “ PUNCT punct
 │ │ ┌── While SCONJ mark
 │ │ ├─┐ much ADJ nsubj
 │ │ │ │ ┌── of ADP case
 │ │ │ │ ├── the DET det
 │ │ │ │ ├── digital ADJ amod
 │ │ │ └─┴ transition NOUN nmod
 │ │ ├── is AUX cop
 │ ├─┴ unprecedented ADJ advcl
 │ │ │ ┌── in ADP case
 │ │ │ ├── the DET det
 │ │ │ ├── United PROPN compound
 │ │ └─┴ States PROPN obl
 │ ├── , PUNCT punct
 │ │ ┌── the DET det
 │ │ ├── peaceful ADJ amod
 │ ├─┼ transition NOUN nsubj
 │ │ │ ┌── of ADP case
 │ │ └─┴ power NOUN nmod
 └─┼ is AUX root
   ├── not ADV advmod
   ├── , PUNCT punct
   ├── ” PUNCT punct
   │   ┌── Obama PROPN compound
   │   ├── special ADJ amod
   │ ┌─┼ assistant NOUN nsubj
   │ │ ├── Kori PROPN flat
   │ │ └── Schulman PROPN flat
   ├─┼ wrote VERB parataxis
   │ │ ┌── in ADP case
   │ │ ├── a DET det
   │ │ ├── blog NOUN compound
   │ ├─┴ post NOUN obl
   │ └── Monday PROPN nmod:tmod
   └── . PUNCT punct
```

A similar output can be generated also as a HTML document (use option -H instead of -T), viewable in any browser.

Udapi blocks

Udapi comes with a number of ready-to-use processing blocks. The documentation contains a list of currently available blocks (https://udapi.readthedocs.io/en/latest/py-modindex.html) with a short description of each block. You can write your own blocks and add them to your copy of Udapi to the folder `udapi-python/udapi/block`. Then they will be visible to Udapi and you can use them to process data. When calling Udapi, you give it as arguments the sequence of blocks you want to apply to your data. For example:

```
cat mydata.conllu | udapy -s ud.FixPunct check_paired_punct_upos=True
ud.FixChain > fixed.conllu
```

The above command lists two blocks to be applied to the trees in this order: `ud.FixPunct` and `ud.FixChain`. Each of the blocks addresses one type of commonly encountered annotation error and tries to fix it automatically. Arguments containing the equals-to symbol (such as `check_paired_punct_upos=True` in the above example) are parameters of the immediately preceding block. The sequence of blocks could start with a reader block that reads data in other format than CoNLL-U; if no such block is listed, Udapi uses `read.Conllu` by default. Similarly, the last block could be a writer block that will write the possibly modified trees to STDOUT or to a file. The option -s we used above means "save" and it is a shortcut that tells Udapi to add the block `write.Conllu`. Similarly, the option -T in the example in the previous section is a shortcut for the block `write.TextModeTrees`. We may want to specify parameters for that writer block, and unless those parameters have their own shortcut, we will have to list the block verbosely:

```
cat en_pud-ud-test.conllu | udapy write.TextModeTrees
attributes='form,lemma,upos,deprel' color=1 | less -R
```

We can also not require any writing of the full trees; perhaps we want to print some statistics or examples directly from other blocks. A simple example of that is the block `util.Wc`, which prints the number of trees and nodes in the input file:

```
cat mydata.conllu | udapy util.Wc
```

The util.Eval block

For simple tasks, you do not have to write a block and save it in a file. You can use a generic block called `util.Eval` (https://udapi.readthedocs.io/en/latest/udapi.block.util.html#module-udapi.block.util.eval) and supply your own code as a parameter. For example, if you provide the parameter "node", its value is interpreted as Python code that will be applied to every tree node encountered in the input file. For example, the following command will read the English PUD treebank, find all nodes whose POS tag (upos) is "AUX" (auxiliary verb or particle). It will print the word form of those nodes converted to all-lowercase (so that capitalized words in the beginning of sentences don't look different) and their lemmas to STDOUT. The subsequent Linux commands will then transform the output to a sorted list with frequencies.

```
udapy util.Eval node='if node.upos == "AUX": print(node.form.lower(),
node.lemma)' < en_pud-ud-test.conllu | sort | uniq -c | sort -rn | less
```

The main node properties that you can access have self-explanatory names derived from the names of corresponding columns in the CoNLL-U file (see also https://udapi.readthedocs.io/en/latest/udapi.core.html#udapi.core.node.Node):

- `form` … actual word form in the sentence
- `lemma` … lemma (base dictionary form)
- `upos` … universal part of speech tag
- `xpos` … treebank-specific part of speech tag (optional)
- `feats` … morphological features stored in an object of the type `DualDict` (https://github.com/udapi/udapi-python/blob/master/udapi/core/dualdict.py). For example, you can query the value of the feature Case like this: `node.feats['Case']`
- `deprel` … dependency relation type/label (pertains to the relation incoming from the parent to this node)
    - `udeprel` … universal part of the dependency relation type. Some treebanks use optional subtypes, e.g., the deprel “acl” (adnominal clause) may have the subtype “acl:relcl” (relative clause). If we want to find all instances of “acl” regardless their subtype, we can ask whether `node.udeprel == 'acl'`. This is equivalent to asking whether `node.deprel.split(':')[0] == 'acl'`.
- `misc` … additional attributes from the MISC column stored in an object of the type `DualDict`. For example, you can ask `if node.misc['SpaceAfter'] == "No"`.
- `ord` … this attribute is not named after a CoNLL-U column, yet it roughly corresponds to the ID column. It is the ordinal numeric value that represents the position of the word in the sentence (the first word has `node.ord == 1`).

Exercise: Find all UPOS tags that co-occur with the deprel “nmod” in your dataset.

Exercise: Find all XPOS tags and their correspondences to UPOS tags. Sort them alphabetically by XPOS tags.

Exercise: Find all UPOS tags that co-occur with a non-empty value of the Case feature and list the Case values they appear with.

In the above list of node attributes that correspond to CoNLL-U columns, we omitted two columns: HEAD and DEPS. The latter encodes the enhanced UD graph and we will ignore it for now. The former encodes the basic tree structure, which in Udapi is accessed via the properties `parent` and `children`:

- `parent` … the Node object of the parent of the current node
- `children` … list of Node objects of the children of the current node, sorted by word order

The following command will find all verbs that have at least two children and print their counts:

```
udapy util.Eval node='if node.upos == "VERB" and len(node.children) >= 2:
print(node.lemma, len(node.children))' < en_pud-ud-test.conllu | sort |
uniq -c | sort -rn | less
```

We can use Python list comprehension to find children with specific properties. Here is a modification of the previous command that will find verbs with two nominal core arguments (their deprel must be “nsubj”, “obj” or “iobj”):

```
udapy util.Eval node='if node.upos == "VERB" and len([x for x in
node.children if x.deprel in ["nsubj", "obj", "iobj"]]) == 2:
print(node.lemma)' < en_pud-ud-test.conllu | sort | uniq -c | sort -rn |
less
```

Exercise: Find nodes whose UPOS tag is “AUX” and the UPOS tag of their parent is not “VERB”. Print the deprel of the parent, then create a sorted list of such deprels with counts.

Exercise: Find inherently reflexive verbs. Czech example: *smát se* is an inherently reflexive verb and it consists of the verbal form *smát* and of the obligatory “reflexive” marker *se*. You will need a language where inherently reflexive verbs exist (i.e., not English; examples include German, or various Slavic and Romance languages). Among such languages, you need a treebank that uses the optional relation subtype “expl:pv” to mark the relation between the verb and its “reflexive” morpheme. List lemmas of the verbs together with the surface forms of their reflexive morphemes; sort them alphabetically. As an English alternative to this exercise, look for phrasal verbs where the verbal particle is attached via a relation labeled “compound:prt”.

Sometimes you want to perform an action for every tree rather than for every node. To do so, use the `tree` parameter instead of `node`. Within the parameter value (the Python code), the identifier `tree` will give you access to the `Node` object of the artificial root of the tree (this node does not correspond to any surface token and the HEAD column of the CoNLL-U file refers to it using the index 0). If you use the property `descendants` of the root, you will get the list of tree nodes sorted by word order.

```
udapy util.Eval tree='print("sentence with", len(tree.descendants),
"nodes")' < en_pud-ud-test.conllu | sort | uniq -c | sort -rn | less
```

Writing your own block

Python syntax relies on line breaks and indentation, which means that it is very unfriendly for squeezing multiple statements on one line. If you need just one `if` statement but multiple commands when the condition is satisfied, you can use semicolons to separate the commands. But if you need multiple conditional branches, you will have to put multiple indented lines inside the single quotes that delimit your `node` parameter:

```
cat en_pud-ud-test.conllu | udapy util.Eval node='if node.deprel ==
"case":
    if len([x for x in node.children if x.udeprel == "fixed"]) >= 1:
        print("FIXED:  ", " ".join([node.form.lower()]+[x.form.lower()
for x in node.children if x.udeprel == "fixed"]))
```

```
    else:
        print("SINGLE: ", node.form.lower())
' | sort | uniq -c | sort -rn | less
```

It is quite messy to do all this directly in the shell, and if there is any chance that you will run the task more than once, you probably want to save the code in a file and use it as a regular block. Fortunately, the blocks can be fairly simple. Examine the folder where you installed Udapi (let's refer to it as `$UDAPI`), take a block, make a copy of it and modify it to suit your needs. For example, `$UDAPI/udapi/block/demo/rehangprepositions.py` is an example of a very simple block that finds a preposition, detaches it from its current parent, attaches it to its current grandparent, then re-attaches the former parent as a child of the preposition. Here is the full code:

```
"""RehangPrepositions demo block."""
from udapi.core.block import Block

class RehangPrepositions(Block):
    """This block takes all prepositions (upos=ADP) and rehangs them
above their parent."""

    def process_node(self, node):
        if node.upos == "ADP":
            origparent = node.parent
            node.parent = origparent.parent
            origparent.parent = node
```

Defining a function called `process_node()` in your block is equivalent to supplying a `node` parameter to the `util.Eval` block: the code of the function is what you would supply as the value of the `node` parameter. Nevertheless, you also must not forget to import the `Block` class and declare your block as a new class derived from the `Block` class. Also note the correspondence between the name of the class and the name of the file.

Let's create a folder for our own blocks, `$UDAPI/udapi/block/my`, and let's copy the RehangPrepositions demo block as `$UDAPI/udapi/block/my/demo.py`. Let's modify the code to just print some info about the current node:

```
"""My demo block."""
from udapi.core.block import Block

class Demo(Block):

    def process_node(self, node):
        cdeprels = ["nsubj", "obj", "iobj"]
        if node.upos == "VERB":
            coreargs = [x for x in node.children if x.deprel in cdeprels]
            if len(coreargs) == 2:
                print(node.lemma)
```

Now you can run Udapi with your new block:

```
udapy my.Demo < en_pud-ud-test.conllu | sort | uniq -c | sort -rn | less
```

Exercise: Find all verbs governing a nominal subject (“nsubj”) and/or one or more objects (“obj”, “iobj”) (plus possibly any number of other children). For each such case, print a line that expresses the order of these elements. In the line, use “V” to represent the verb, and use the deprels to represent the arguments. For example: “nsubj V obj”. Do not print other children of the verb (like adverbial modifiers). If there are multiple objects, print them all. Count the distribution of the word orders and list them sorted by frequencies.

Location of the blocks on the disk

In the above example we created our own block directly in the folder where we have installed our copy of Udapi: `$UDAPI/udapi/block/my/demo.py`. This is the simplest way of making sure that Python will find the block. When we tell Udapi to use a block, e.g., “`my.Demo`”, Udapi will lowercase it, change periods to slashes and prepend “`udapi/block/`”, that is, it will tell Python to look for the module “`udapi/block/my/demo`”. Python will then search the paths from the `$PYTHONPATH` variable for this module.

This may not exactly be how we want to organize our code. Maybe we want to treat Udapi as an external tool and keep our own blocks separate from it, but bundled and versioned with our own project. Creating our own folder “`udapi/block`” and adding it to `$PYTHONPATH` is not recommended. Python has been known to get confused when there are multiple instances of “`udapi/block`” in `$PYTHONPATH`, and to only search the first instance and ignore the subsequent ones. However, you can insert a dot at the beginning when giving the block name to Udapi; this will signal that Udapi should not prepend “`udapi/block/`” to the block name. Then you can simply have the block in your project folder and add that to `$PYTHONPATH` (or keep it in the current working folder, which will be searched by Python, too).

```
udapy .my.Demo < en_pud-ud-test.conllu | sort | uniq -c | sort -rn | less
```

Using Udapi as a library from your Python script

If you need to do something very specific (i.e., not worth being implemented as a reusable block) and/or you need to process CoNLL-U files in your own Python project (perhaps surrounded by other code), you can import the Udapi modules in your script.

The easiest way of using the Udapi interface to CoNLL-U files is to import the `Document` class. Note an important terminological detail: a “document” here refers to the entire contents of one CoNLL-U file. Some UD files contain metadata lines (starting “`# newdoc`”) that indicate boundaries of separate logical documents within the underlying text corpus. Udapi even has some limited support for work with these boundaries, if they are present. But these logical documents are not what the `Document` class refers to.

```
"""A script that processes a CoNLL-U file. It adds VerbForm=Fin to the
features of every verb that has no VerbForm so far. The modified file
```

```
will be saved as output.conllu, but the script also prints sentence ids
and forms of the modified verbs to STDOUT."""
from udapi.core.document import Document

# Read the CoNLL-U file.
document = Document(filename='in.conllu')
# Examine every node in every tree and do something with it.
for b in document.bundles:
    root = b.get_tree()
    print(root.sent_id)
    nodes = root.descendants
    for node in nodes:
        if node.upos == 'VERB' and node.feats['VerbForm'] == '':
            print(node.form)
            node.feats['VerbForm'] = 'Fin'
# Write the modified CoNLL-U file.
document.store_conllu('out.conllu')
```

You can also run an existing Udapi block from your code. If you know that the only functionality of the block is implemented in its `process_node()` method, you can call that method as you loop over the nodes in the document, and combine it with your additional code if needed. Same for `process_tree()`, `process_bundle()` and `process_document()`. However, you should know that some blocks also implement initialization and finalization methods, such as `before_process_document()`, `after_process_document()`, `process_start()` (to be invoked once before processing any documents) and `process_end()`. You are responsible for calling these methods at appropriate places, too. If you do not need to add your own code in the loop, you can rely on Udapi to construct the loop for you, and call either `block.run(document)` (calls all pre- and postprocessing methods) or `block.apply_on_document(document)` (calls `before/after_process_document()` but not `process_start/end()`).

```
from udapi.core.document import Document
from udapi.block.ud.fixpunct import FixPunct

document = Document(filename='in.conllu')
# We can add parameters to the block that we would otherwise add on the
command line.
fixpunct = FixPunct(check_paired_punct_upos=True)
fixpunct.run(document)
document.store_conllu('out.conllu')
```

The blocks util.Filter and util.Mark

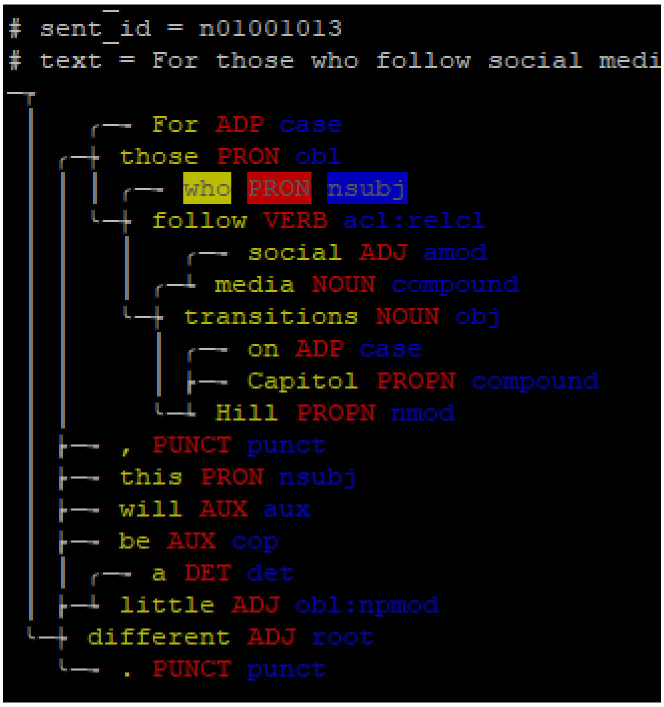
Another useful block that comes with Udapi is `util.Filter` (https://udapi.readthedocs.io/en/latest/udapi.block.util.html#module-udapi.block.util.filter). As the name suggests, it filters the trees from the input, i.e., subsequent blocks will only see trees that fulfill a constraint. In the following example, the parameter `keep_tree_if_node` provides a

Python expression that evaluates to a `True|False` value. If the current tree contains at least one node for which the expression evaluates to `True`, the tree is kept; otherwise it is discarded. Our condition is a regular expression match: the feature "PronType" must contain either the string "Rel" (relative) or the string "Int" (interrogative). We use the option `-T`, hence the trees that survive will be rendered using text characters and sent to STDOUT. This time we also use the `-N` option (equivalent: `--no_color`) because we want to save a text file instead of looking at the trees directly in the terminal, and we want to skip the color control characters. Run `udapy --help` to discover other command-line options.

```
udapy -TN util.Filter keep_tree_if_node='re.match("Rel|Int",
node.feats["PronType"])' < en_pud-ud-test.conllu > rel-int-trees.txt
```

Optionally, we can also add a `mark` parameter. Its value will be used as a label that will be added to the MISC column of the node where the `_if_node` condition is met. For instance, if we add the parameter `mark=here`, the MISC column of the given node will contain the attribute "Mark=here". It can be used by subsequent blocks and it can be saved in the CoNLL-U file if Udapi is called with the `-s` option. If we call Udapi with the `-T` option, the nodes containing a Mark attribute will be highlighted:

```
udapy -T util.Filter keep_tree_if_node='re.match("Rel|Int",
node.feats["PronType"])' mark=here < en_pud-ud-test.conllu | less -R
```



Note: There is also a block called `util.Mark` (https://udapi.readthedocs.io/en/latest/udapi.block.util.html#module-udapi.block.util.mark), which marks nodes that fulfill a condition but does not remove trees in which no node is marked:

```
udapy -T util.Mark node='re.match("Rel|Int", node.feats["PronType"])'
add=False < en_pud-ud-test.conllu | less -R
```

Exercise: Find nodes whose dependency on their parent is non-projective, mark those nodes and display the respective trees in the textual form. Hint: You do not have to implement the condition whether all nodes between the node and its parent are descendants of the parent. It has been already implemented. All you have to do is to use the method `is_nonprojective()` of the `Node` object.

The block provides multiple ways of specifying the filtering constraint:

- `keep_tree_if_node` … see above. Every node is examined individually but any node fulfilling the constraint will make the whole tree survive.
- `delete_tree_if_node` … complementary constraint: any node fulfilling the condition will cause the tree to disappear.
- `keep_tree` … we specify a condition for the whole tree rather than for an individual node.
- `delete_tree` … we specify a negative condition for the whole tree.
- `keep_subtree` … we specify a condition for a node; the node and its subtree (all its descendants) will survive if the condition evaluates to `True`. If there are multiple surviving subtrees within one original tree, each of them will be attached directly to the artificial root node (note that this diverges from the UD treebanks where there is always just one node attached as child of the artificial root). If no node fulfills the condition, the entire tree will be discarded.
- `delete_subtree` … removes a node and its descendants if the condition evaluates to `True`.
- `keep_node` … only nodes fulfilling the condition will be kept. If a node's parent is removed, the node will be re-attached to the next available ancestor. If no node fulfills the condition, the entire tree will be removed.

Exercise: Remove punctuation (UPOS = "PUNCT") from all trees. Find trees that, after removing punctuation, have at least 5 but no more than 15 nodes. Among these trees, mark each node whose form contains an uppercase letter but the node is not the first word of the sentence and its UPOS tag is not "PROPN" (proper noun); keep only trees that contain at least one such node.

Exercise: Same as the previous one but instead of printing the filtered trees, count their number and the number of words in them.

Exercise: Filter a corpus so that only trees whose sentence id does not start with "vesm" are preserved. (You can try this with UD_Czech-PDT where "vesm" denotes one of the four newspaper sources of the corpus text.) Hint: You can query `tree.sent_id`.

Exercise: Filter a corpus so that only trees whose sentence-level comments contain the comment "Tectogrammatical annotation available." are preserved (this, too, holds for some but not all sentences in UD_Czech-PDT). Hint: You can query `tree.comment.split("\n")`. All the sentence-level comment lines are available in this attribute in their original order, delimited by line breaks. The comment-signalizing character "#" is not included but everything else (including leading spaces) is. This is only useful for comments that have no special meaning for Udapi and are not consumed by Udapi and presented as separate tree attributes. For comments that are expected and consumed by Udapi, the respective comment line will contain only a placeholder such as `$SENT_ID`, `$TEXT`, `$NEWDOC`, `$NEWPAR`, `$GLOBAL.ENTITY`.

Navigating the nodes in the tree

As we have seen, Udapi allows us to easily access the parent and the children of a node in the tree. There are more methods (or properties) that help us navigate the tree both vertically (parent-child relations) and horizontally (word order). Here is an overview; see https://udapi.readthedocs.io/en/latest/udapi.core.html#udapi.core.node.Node for details.

- `node.root` … returns the node object of the artificial root of the tree; this is not the word with the "root" deprel but its parent!
- `node.parent` … returns the parent node object
- `node.children` … returns the list of children sorted by word order. You can request adding the current node to the sorted list. You can also restrict the list to left children or right children of the current node:
    - `phrase = node.children(add_self = True)`
    - `lchildren = node.children(preceding_only = True)`
    - `rchildren = node.children(following_only = True)`
    - `selfleft = node.children(preceding_only = True, add_self = True)`
- `node.descendants` … returns the list of descendants sorted by word order. Parentheses with the same modifiers can be added as for `node.children`.
- `node.siblings` … returns the list of siblings of the current node, sorted by word order. Parentheses with the same modifiers can be added as for `node.children`.
- `node.prev_node` … returns the preceding node according to word order (or `None` if this is the first node).
- `node.next_node` … returns the following node according to word order (or `None` if this is the last node).

Exercise: In a Spanish corpus, find all examples of subordinate clauses that start *el que* "the one that", optionally with a preposition, such as *sobre el que* "about the one that". The determiner *el* can be in various forms (masculine, feminine, singular, plural); find all whose lemma is "el". You can assume that *que* is attached as a child of a predicate (typically a verb) but you do not know how *el* and the preposition are attached (perhaps it is inconsistent and needs to be fixed). However, you know that these words immediately precede *que* in the sentence. (In contrast, the verbal parent of *que* is not necessarily adjacent to *que*.) For each instance found, print the structure of the construction in Stanford notation, i.e., deprel(parent word, child word), for example `nsubj(corre, que)`. Use "ADP" to represent the preposition, "el" to represent the determiner *el* (*la*, *los*, *las*), “que” to represent the relative pronoun *que*, “VERB” to represent the predicate that *que* is attached to, “OTHER” to represent any other node that *el* or the preposition or the verb may be attached to. So the full Stanford-style description of the structure may be, e.g., “`case(el, ADP); root(OTHER, el); nsubj(VERB, que); acl:relcl(el, VERB)`” (relations are sorted by word order of the child nodes). Summarize all patterns observed and print them with their frequencies, most frequent pattern first.

Exercise: In any corpus that uses the `fixed` relation, collect all types of fixed multiword expressions (two or more words connected with `fixed` relations, with the leftmost word as the technical head). Then search the corpus again and see if any of these expressions also occur with different relations than `fixed`.

Reading large files

By default, Udapi reads the entire input file into memory before processing it. This may be a problem if the input is too large. If we know that we only need to look at one sentence at a time, we can instruct Udapi to split the input into small documents, each consisting of one “bundle” (sentence). That way we will not have to wait until the whole input is read before we see the first results, and we will not risk running out of memory. To achieve this, the reader block must receive the parameter `bundles_per_doc=1`.

```
zcat huge.conllu.gz | udapy -T read.Conllu bundles_per_doc=1 util.Mark
node='re.match("Rel|Int", node.feats["PronType"])' add=False | less -R
```

Modifying the data

So far we have mostly focused on getting information from the data, without modifying the data. (But strictly speaking, we did modify the data occasionally. Sometimes we filtered the trees or added marks to nodes. If we then saved the CoNLL-U file instead of just displaying the trees, we would have a modified corpus.)

Udapi is often used to automatically modify UD data following specific rules or heuristics. The modification can consist of changing tags or features of individual nodes, but also of transforming the tree structure.

As with collecting information, you can either write your own block, or, if the modification is simple, you can specify it as a parameter to the `util.Eval` block.

To demonstrate a potentially useful modification, we will now turn to data in languages other than English, written in a non-Latin alphabet. UD defines two MISC attributes that can help non-native users to read the words and lemmas: “Translit” and “LTranslit”. These attributes are completely optional (as almost everything in MISC) but some treebanks have it. In addition, some treebanks also have the “Gloss” attribute, which provides a translation of the word (usually English translation). Now if you are looking at trees via `udapy -T` and you cannot read the script used by the language, you may prefer to see the transliteration in the FORM and LEMMA fields. And if the script is written right-to-left, you may prefer to see the transliteration even when you can read the script, because otherwise the terminal messes up the text on the line. Consider the following example from the beginning of the training data of UD_Arabic-PADT:

```
cat ar_padt-ud-train.conllu | udapy -T | less -R
```

When the first letter encountered on a line is Arabic, the terminal renders the line right-to-left, but then switches to left-to-right when Latin letters are encountered. The two lines with quotation marks as tokens are entirely left-to-right. In either case, the tree structure is obscured. So let's modify the data before we display it. Let's replace the word form by its transliteration from MISC (fortunately, Arabic PADT is one of the treebanks that have it).

```
cat ar_padt-ud-train.conllu | udapy -T util.Eval node='if
node.misc["Translit"] != "": node.form = node.misc["Translit"]' | less -R
```

Moreover, PADT also has English glosses, so we can even get a rough idea of the meaning of the individual words:

```
cat ar_padt-ud-train.conllu | udapy -T util.Eval node='if
node.misc["Translit"] != "": node.form = node.misc["Translit"] + " (" +
node.misc["Gloss"] + ")"' | less -R
```

```
docname = afp.20000715.0075
loaded_from = -
# sent_id = afp.20000715.0075:plul
# text = برلين ترفض حصول شركة اميركية على رخصة تصنيع دبابة "ليوبارد" الا لمانية
─┬
 │ ┌── barlīn (Berlin) X nsubj
 └─┼ tarfuḍu (reject,refuse) VERB root
   └─┬ ḥuṣūla (acquisition,obtaining,occurrence,happening) NOUN obj
     ├─┬ šarikatin (company,corporation) NOUN nmod
     │ └── ʾamīrikīyatin (American) ADJ amod
     │ ┌── ʿalā (on,above) ADP case
     └─┼ ruḫṣati (license,permit) NOUN obl:arg
       └─┬ taṣnīʿi (fabrication,industrialization,processing) NOUN nmod
         └─┬ dabbābati (tank) NOUN nmod
           │ ┌── " () PUNCT punct
           ├─┼ liyūbārd (Leopard) X nmod
           │ └── " () PUNCT punct
           └── al-ʾalmānīyati (German) ADJ amod

:
```

How to split a sentence into two

If we spot a sentence that is in fact two sentences and needs to be split, we do not want to do it with manual editing, even if it is just a single split operation, because it would be tedious: All nodes in the second part and all references to them must be renumbered. Fortunately, there is now a block that can do it for us. You should still check the result, as there may be multiple root relations (if there were multiple relations between the first and the second part), which is not allowed in UD. Also be warned that the current version of the block does not take care of enhanced dependencies if they are present. The word_id parameter is the old ID of the first node of the second part of the sentence.

```
cat hi_hdtb-ud-train.conllu | udapy -s util.SplitSentence sent_id='train-s842' word_id=7 > hi_hdtb-ud-train-split842.conllu
```

How to split a node (word, token) into two

Suppose the treebank uses different tokenization rules than you prefer. For example, *20%* is one node and you want it always split into two nodes, the number *20* and the symbol *%*. You can use a regular expression to identify all such nodes. When you find one, you need to create a new node, give it the correct position in the tree, and move part of the original word to it. New nodes are always created as children of existing nodes, using the parent node's method `create_child()`. The default linear position of a new node is at the end of the sentence but we can re-position it using the methods `shift_before_node()` and `shift_after_node()`, which will also take care of renumbering the ids of the other nodes.

```
def process_node(self, node):
    m = re.match(r'^([0-9]+(?:[\.,][0-9]+)?)%$', node.form)
    if m:
        number = m.group(1)
```

```
        numbernode = node.create_child()
        numbernode.shift_before_node(node)
        numbernode.form = number
        numbernode.misc['SpaceAfter'] = 'No'
        # Make the lemma same as the form.
        # If the original lemma normalizes decimal comma to point,
        # we may want to use regex and split the orig lemma instead.
        numbernode.lemma = number
        numbernode.upos = 'NUM'
        numbernode.feats['NumType'] = 'Card'
        numbernode.feats['NumForm'] = 'Digit'
        numbernode.deprel = 'nummod'
        node.form = '%'
        node.lemma = '%'
        node.upos = 'SYM'
        # If the original node had any features, remove them.
        node.feats = {}
```

Important notes: The above code does not assume that the original node can be a part of a multi-word token (see below), which might require additional steps to make sure that the newly created node is also part of that multi-word token. Furthermore, if there is an enhanced graph for the sentence (see below), the attachment of the new node to the enhanced graph must be taken care of.

Exercise: Find all nodes that correspond to ranges of positive integer numbers (such as years: *1968-1971*). When you find such a node, split it to three: each number will have its own node, and the hyphen will be separate, too. Treat the range as coordination, i.e., the second number is attached to the first one as `conj`, and the hyphen is attached to the second number as `punct`.

Multi-word tokens

A specialty of UD and its CoNLL-U file format is that it can work with syntactic words (nodes) that correspond only to a part of an orthographic word (surface token). For example, the Spanish word *al* is a contraction of the preposition *a* and the definite article *el*. A CoNLL-U file has an extra line describing the surface word (multi-word token, MWT) *al*, showing only its ID, FORM, and MISC (the other columns have underscores in them). The line is followed by normal lines corresponding to the nodes *a* and *el*. When processing a Spanish UD tree in Udapi, you will see the nodes corresponding to *a* and *el* but you will normally not encounter the MWT they are part of—the MWT is not a node! However, if you need to work with the MWT, you can access it from any node that belongs to it. MWTs in Udapi are objects of the class `MWT` (see [udapi.core.mwt]) and their attributes are `form`, `misc`, `ord_range`, and `words` (a list of node objects that belong to the MWT).

Here is how we can list all multi-word tokens in a UD treebank:

```
cat es_pud-ud-test.conllu | udapy util.Eval node='if node.multiword_token and (node == node.multiword_token.words[0]): print(node.multiword_token.form, "=", [x.form for x in node.multiword_token.words])' | sort | uniq -c | sort -rn | less
```

Exercise: The MISC column in CoNLL-U may contain the attribute "SpaceAfter=No", which indicates that in the original untokenized text there was no space between this and the following token. In case of a multi-word token, this attribute must occur on the line describing the MWT rather than its last (or any other) node. If it occurs at a node within a MWT, the UD validator will report it as an error. Your task is to look for such errors in the data and fix them, i.e., remove the attribute from the MISC of the node and, if it was the last node of the MWT, add it to the MISC of the MWT.

Enhanced UD

If the treebank includes enhanced dependencies, the CoNLL-U file has two parallel structures for each sentence: the basic tree and the enhanced graph. Everything we were doing so far pertained to the basic tree. The enhanced graph structure is stored in the DEPS column of the CoNLL-U file. Essentially, when looking at the DEPS field on the line of a particular word (node), we see a list of incoming edges from all parents of the current node in the enhanced graph. Each edge consists of two components: the identification of the parent node, and the label (type) of the edge.

If you access the `node.deps` property, you should receive a list of Python dictionaries, each with two elements: `parent` and `deprel`. The former should be a `Node` object. Here is how we could remove the enhanced annotation from the file:

```
udapy -s util.Eval node='node.deps = []' < en_pud-ud-test.conllu >
without_enhanced.conllu
```

The following command will copy the enhanced dependency type to the basic tree, provided there is only one enhanced parent, it is identical to the basic parent, and the enhanced dependency label does not violate the rules for basic dependency relations.

```
udapy -s util.Eval node='if not node.is_empty() and (len(node.deps) == 1)
and (node.deps[0]["parent"] == node.parent) and (node.deps[0]["deprel"] !
= node.deprel) and (re.match(r"[a-z]+(:[a-z]+)?$", node.deps[0]
["deprel"])): node.deprel = node.deps[0]["deprel"]' < input.conllu >
output.conllu
```

The following command will find and show examples of the English enhanced relation “obl:with”.

```
udapy -T util.Filter mark=here keep_tree_if_node='len(node.deps)>=1 and
"obl:with" in [x["deprel"] for x in node.deps]' < input.conllu | less -R
```

Important: The method `process_node()` (and, by extension, the `node` attribute of util.Eval) works only with the regular nodes of the basic tree. It skips empty nodes, which are part of the enhanced graph but not of the basic tree. If we want to visit all nodes of the enhanced graph, we have to take a different approach. We have to use the method `process_tree()`, then at the root of each tree/graph we get the property `root.descendants_and_empty`, which will return a list of nodes. We can then loop over that list and do for each node what we need to do. If we want to visit only empty nodes, we can use the property `root.empty_nodes` instead.

At present (still in July 2023), there is a bug in Udapi: the methods `node.shift_after_node()` and `shift_before_node()` will not update enhanced relations in the DEPS column, nor the empty node ids. See issue https://github.com/udapi/udapi-python/issues/95.

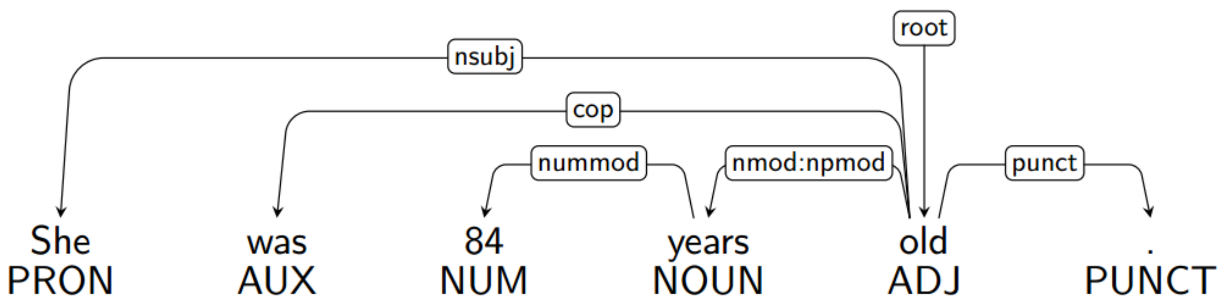
Tree visualization in LaTeX

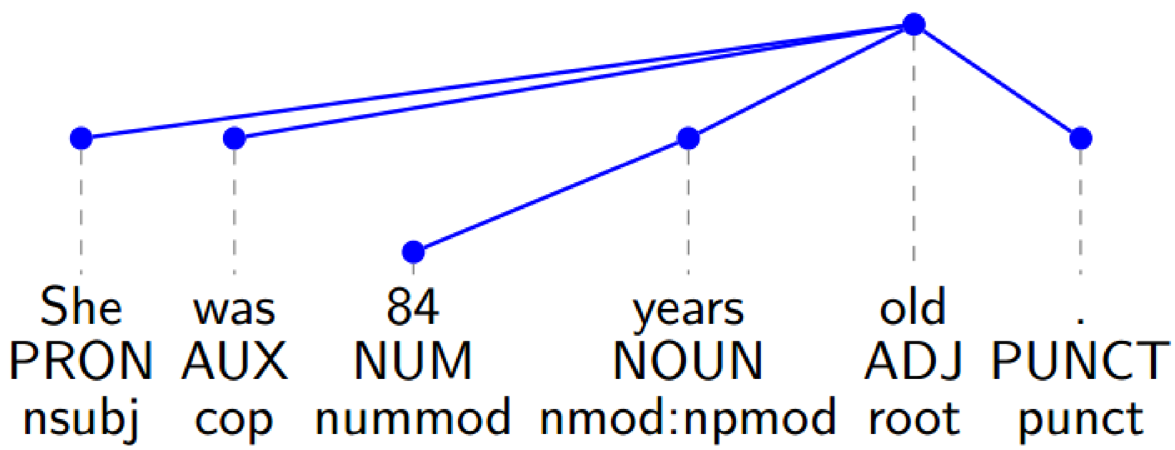
LaTeX is a typesetting system used to write conference papers, diploma theses, or even presentation slides. While there are multiple ways of showing dependency trees in LaTeX, the most popular way seems to be the package called tikz-dependency. Udapi can print trees in the format required by tikz-dependency, so you can easily find examples in the treebank and immediately insert them in your article. Just use the block `write.Tikz` (https://udapi.readthedocs.io/en/latest/udapi.block.write.html#module-udapi.block.write.tikz):

```
cat en_pud-ud-test.conllu | udapy util.Filter
keep_tree='len(tree.descendants) == 6' write.Tikz > examples.tex
```

```
pdflatex examples.tex
```

```
xdg-open examples.pdf
```





Homework 4

Morphological features are an optional part of UD annotation, although all data providers are strongly encouraged to provide them. If a treebank has features, it should provide a non-empty value of the "PronType" feature for every pronoun (UPOS tag "PRON") and determiner (UPOS tag "DET"). In addition, there are probably also pronominal adverbs such as *where* and *when*; these should have the “PronType” feature but regular adverbs do not have it. For documentation of the feature, see https://universaldependencies.org/u/feat/PronType.html.

Your task is to find a treebank in the UD release 2.11 where some or all pronouns or determiners lack the “PronType” feature. Collect all lemmas (or word forms, if the treebank lacks lemmas) of pronouns and determiners in the treebank. Write a block that will examine each pronoun/determiner (and optionally also adverb) node. If it lacks the “PronType” feature, the block decides, based on its lemma (or word form) what value of “PronType” it should have, and adds the value to the features of the node.

Submission: Upload the block to your folder in the SVN repository. Name the block “`addprontype.py`”. After uploading the block, send me (zeman at ufal) a message that your block is ready for review; also tell me in the message which language your block works with and which treebank of the language you used for testing. Deadline: TBA, 23:59 CEST.