Re-generating sentences from Universal Dependencies Structures

Guy Lapalme
RALI-DIRO, Université de Montréal
lapalme@iro.umontreal.ca

December 28, 2020

Abstract

We describe a system for regenerating sentences from English or French Universal Dependencies structures (UD). A symbolic approach is used to transform the dependency tree into a tree of constituents which is then be (re)realized as a sentence in the original language. We show how the output of the system can be used as a validation tool for UDs.

Context of the work

We started to work on Universal Dependencies (UD) structures in the context of the Second Workshop on Multilingual Surface Realization [4] which used transformed Universal Dependencies structures as input to a surface realizer. This realizer [3] used Prolog for parsing the input structures from which a constituent tree structure was built and sent to jsRealB, a JavaScript English realizer that we have been developing for some time. The approach we followed was to first build a realizer for the original Universal Dependencies and then adapt it to take into account the transformations used for the competition: i.e. scrambling of words and abstracting relations keeping only content words.

Alessandro Sordoni, who was aware of our work asked if it was possible to use the UDs produced by a statistical parser (Stanza [7]) of an affirmative sentence to produce automatically its negative form. These modified sentences would then be used as a training corpus to *teach* the meaning of negation to a neural language model. This system also used Prolog for parsing and creating a JSON-based tree constituent structure that was interpreted by jsRealB.

We now describe UDregenerator which uses a slightly different approach in which all the transformation is performed in JavaScript: parsing of Universal Dependencies, transformation and text generation. The whole approach is now integrated in a web page (see

Figure 1) which allows entering Universal Dependencies structures, re-generation of the sentence from the information in the dependencies, comparison of the realized sentence with the original. The dependencies can be modified and (re)parsed. The transformed constituent expression can also be edited for regenerating the sentence.

UDREGENERATOR can also be used as a NODE.JS module for batch processing a UD dependency file. One further motivation for this work was to study the coverage of JSREALB or English and French, so we decided to sample UDs in these language to check to what extent it was possible to recreate verbatim the original sentences. This experiment also allows to measure to what extent the lexical information in UD is exact or complete.

We first recall the UD input format and describe the tree-based representations used by our system using the example from Figure 1. Section 3 presents the core algorithm for transforming between the these representations. Section 4 describe our experience in using UDREGENERATOR for validating UDs. We conclude with some lessons learned from this development.

1 Universal Dependencies

Universal Dependencies structures [6] (UD) is an open community effort to create cross-linguistically consistent treebank annotation for many languages within a dependency-based lexicalist framework. The latest version (2.7) [9] provides 183 treebanks in 104 languages. This data has been developed for comparative linguistics and is used in many NLP projects for developing parsers, considering UDs as gold standard.

The UD structures are provided in tab separated files in a systematic format[1]. A UD annotation is a series of lines with the following 10 fields, an empty field is indicated by an underscore (_).

`id`	word index, starting at 1
`form`	how the token is written out
`lemma`	the lemma of the `FORM`
`upos`	*universal* part of speech tag of word
`xpos`	language specific part of speech (ignored here)
`feats`	list of morphological features (abbreviated here)
`head`	`ID` of the head or 0 for the head
`deprel`	name of the *universal* dependency relation to the `HEAD`
`deps`	*enhanced* dependency graph (not used in our work)
`misc`	supplementary annotation, such a spacing before and after the token

[1] `https://universaldependencies.org/format.html`

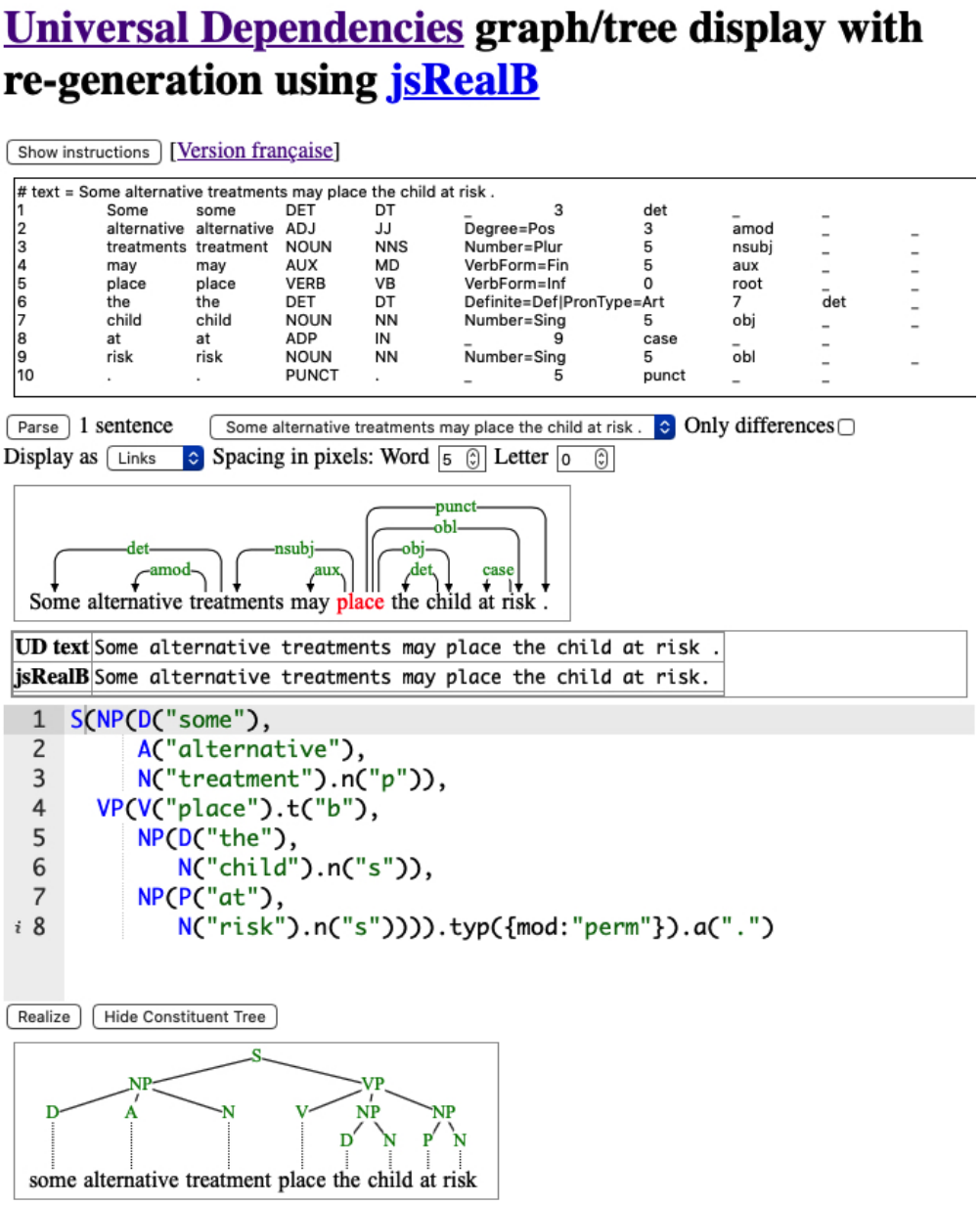


Figure 1: Web page (`http://rali.iro.umontreal.ca/JSrealB/current/demos/UDregenerator/UDregenerator-en.html`) for building jsRealB expressions from a list UDs given in the text area at the top. A menu allows the selection of a sentence for which the dependency structure is displayed as well as a jsRealB expression created in the editor area at the bottom. The realization is shown in the middle row of the table above the editing area. When there are differences between the expected text and realized text, they are highlighted. It is then possible to either correct the jsRealB expression or the UD dependency and to either re-parse the dependencies or re-realize the expression. The tree of constituents corresponding to the jsRealB expression is displayed at the bottom.

Comments are added to the file using lines with a sharp sign (#). There are conventional comments such as: a line starting with `# id =` uniquely identifies a dependency structure in a file and a line starting with `# text =` indicates the text of the sentence for which the dependency structure is defined. A file can contain many UDs that are separated by a an empty line.

Many of these dependency structures are the result of manual revisions of automatic parses which are often quite difficult to check manually as there so many details to take into account. As we will show in Section 4, regenerating from the source revealed small mistakes in quite a few of the structures. It is indeed much easier to detect errors in a drawing or in a generated sentence than in list of tab separated lines.

2 UDregenerator

Figure 1 shows the web based interface of UDREGENERATOR using a simple English sentence. The text area at the top shows the UD input with the corresponding dependency link structure in the middle.

The original UD structure is parsed to build a dependency tree which is then converted to a tree of constituents realized using JSREALB[2], a web-based English and French realizer written in Javascript. Only the English realizer is illustrated here but there is also a web page for using the system dealing with the French version of UD. The bottom of Figure 1 shows the tree of constituents built by JSREALB after processing the UD.

An UD realizer might seem pointless, because most UD annotations are created from realized sentences either manually or automatically. As UDs contains all the tokens in their original form (except for elision in some cases), the realization can be obtained trivially by listing the `form` in the second column of each line.

Taking into account the tree structure, another baseline generator can be implemented using an in-order traversal of the tree and output the `form`s encountered. This method does not work for *non-projective* dependencies [2] because words, in this case, under a node are not necessarily contiguous. We use this property in our system to detect non projective dependencies which account for about 5% of the dependencies in our corpora. But even for projective ones, different trees can be linearized in the same way. But quite often, non-projective dependencies are a symptom of badly linked nodes that should be checked.

What we propose in this paper is UDREGENERATOR that uses only the lemmas and the morphological and syntactic information contained in the UD features and relations to realize a sentence *from scratch* which can be compared to the original. Interesting use cases for such a realizer could be:

- Should a *What to say* module of an NLG system produce UD structures, then UDREGENERATOR could be used as the *How to say* module.
- An UD structure obtained by an automatic parser can be used to create variations of the original sentence using JSREALB.

[2] `http://rali.iro.umontreal.ca/rali/?q=en/jsrealb-bilingual-text-realiser`

- Providing help to annotators to check if the information they entered is correct by regenerating the sentence from the dependencies. This enables to catch more types of errors in the annotation; this is not foolproof, but Section 4 describes our experience with this use-case.

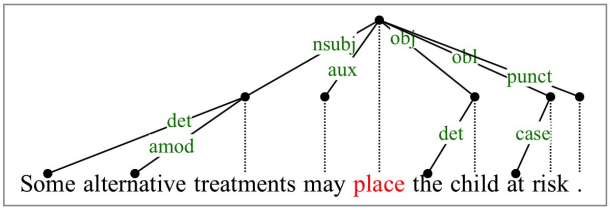
2.1 UD in JSON

The first step in UDREGENERATOR is to parse a group of lines in CONLLU format corresponding to UD structure in order to build the corresponding tree using node objects that keep track of the values of the fields. Once all nodes have been created, they are linked back to their parent using the `head` field, the root being identified by 0 in its `head` field. The features are transformed into an object to ease checking their values.

```
{deprel , upos , lemma , form , id , feats ,
 list of left children ,
 list of right children}
```

This representation keeps intact the parent-child relations and the relative ordering between the children, it also keeps track of the fact that some children occur to the left or to the right of the parent. This is easily inferred from the `ID` of each token compared with the value of its `HEAD`.

This link structure corresponds to a tree structure (see Figure 2) defined using the `head` field that refers to the `id` of the parent. This tree structure can also be displayed in the web page by selecting in the menu in the middle of the page.




```
{"deprel":"root", "upos":"VERB", "lemma":"place", "form":"place", "id":5, "head":0,
 "feats":{"VerbForm":"Inf"},
 "left":[{"deprel":"nsubj", "upos":"NOUN", "lemma":"treatment", "form":"treatments", "id":3, "head":5,
          "feats":{"Number":"Plur"},
          "left":[{"deprel":"det", "upos":"DET", "lemma":"some", "form":"Some", "id":1, "head":3},
                  {"deprel":"amod", "upos":"ADJ", "lemma":"alternative", "form":"alternative", "id":2, "head":3,
                   "feats":{"Degree":"Pos"}}]},
         {"deprel":"aux", "upos":"AUX", "lemma":"may", "form":"may", "id":4, "head":5,
          "feats":{"VerbForm":"Fin"}}],
 "right":[{"deprel":"obj", "upos":"NOUN", "lemma":"child", "form":"child", "id":7, "head":5,
           "feats":{"Number":"Sing"},
           "left":[{"deprel":"det", "upos":"DET", "lemma":"the", "form":"the", "id":6, "head":7,
                    "feats":{"Definite":"Def","PronType":"Art"}}]},
          {"deprel":"obl", "upos":"NOUN", "lemma":"risk", "form":"risk", "id":9, "head":5,
                    "feats":{"Number":"Sing"},
           "left":[{"deprel":"case", "upos":"ADP", "lemma":"at", "form":"at", "id":8, "head":9}]},
          {"deprel":"punct", "upos":"PUNCT", "lemma":".", "form":".", "id":10, "head":5}]}
```


Figure 2: Tree structure extracted from the dependency structure of Figure 1 and its corresponding JSON representation.

2.2 jsRealB

jsRealB[5] is a surface realizer written in Javascript similar in principle to SimpleNLG [1] in which programming language instructions create data structures corresponding to the constituents of the sentence to be produced. Once the data structure (a tree) is built in memory, it is traversed to produce the list of tokens of the sentence.

The data structure is built by function calls whose names were chosen to be similar to the symbols typically used for constituent syntax trees[3]:

- **Terminal**: `N` (Noun), `V` (Verb), `A` (adjective), `D` (determiner) ...
- **Phrase**: `S` (Sentence), `NP` (Noun Phrase), `VP` (Verb Phrase) ...

Usually in Javascript, identifiers starting with a capital letter are constructors not functions, however in linguistics, symbols for constituents start with a capital letter, so we kept this convention. Features added to the structures using the dot notation, called options, can modify their properties. For terminals, their person, number, gender can be specified. For phrases, the sentence may be negated or set to a passive mode; a noun phrase can be pronominalized. Punctuation signs and HTML tags can also be added.

For example, in the jsRealB structure of Figure 1, plural of `treatment` is indicated with the option `n("p")` where `n` indicates number and `"p"` plural. Agreements within the `NP` and between `NP` and `VP` are performed automatically, although this feature is not used often in this experiment because features on each token provide, in principle, all the necessary morphological information . The affirmative sentence is modified to use the *permission* modal using the property `{typ({"mod":"perm"})` to be realized by the verb `may`. The modification of a sentence structure is an interesting feature of jsRealB. Once the sentence structure has been built, many variations can be obtained by simply adding a set of options to the sentences, to get negative, progressive, passive, modality and some type of questions. It would be possible to get the negative sentence, `typ({"mod":"perm",neg:true})` without changing the original jsRealB expression, but this is not studied in detail in this paper.

3 Building the Syntactic Representation

We now describe how a UD in JSON is transformed into a Syntactic Representation (SR) which is used as input to jsRealB. The principle is to *reverse engineer* the universal dependencies annotation guidelines[4]. This is similar to the method described by Xia and Palmer [8] to recover the syntactic categories that are *projected* from the dependents and to determine the extents of those projections and their interconnections.

Although this projection process is theoretically simple, there are some peculiarities when it must be applied in practice between two predefined formalisms for which the idiosyncrasies

[3]See the documentation `http://rali.iro.umontreal.ca/JSrealB/current/documentation/user.html?lang=en` for the complete list of functions and parameter types.

[4]`https://universaldependencies.org/guidelines.html`

must be taken into account. In our case, the specifics of UD relations with features being associated with each word. They must be mapped into jsRealB constituents with options that are applied either to a terminal or a phrase. We now give more detail on the mapping process.

3.1 Morphology

Terminals in UD are objects whose left and right lists of children are empty. They are mapped to *terminal* symbols in jsRealB. So we transform the UD notation to the SR one by mapping lemma and feature names. The following table gives a few examples:

JSON fields	SR
`"upos":"NOUN", "lemma":"treatment", "feats":{"Number":"Plur"}`	`N("treatment").n("p")`
`"upos":"VERB", "lemma":"lean", "feats":{"Mood":"Ind","Tense":"Past"}`	`V("lean").t("ps")`
`"upos":"PRON", "lemma":"its", "feats":{"Gender":"Neut","Number":"Sing", "Person":"3","Poss":"Yes","PronType":"Prs"}`	`Pro("me").c("gen").pe("3") .g("n").n("s")`

As shown in the last example, we had to *normalize* pronouns to what jsRealB considers as its base form. In the morphology principles of UD[5], it is specified that *treebanks have considerable leeway in interpreting what "canonical or base form" means.* In some English UD corpora, the `lemma` of a pronoun is almost always the same as its `form`; it would have been better to use the tonic form. We decided to *lemmatize further* instead of merely copying the lemma as a string input to jsRealB so that verb agreement could eventually be performed. English UDs does not seem to have a systematic encoding of possessive determiners such as `his` which, for jsRealB at least, should be pos-tagged as a possessive determiner. These are defined as pronouns in some sentence or determiners in others, we found even cases of both encodings occurring in the same sentence. As the documentation seems to favor pronouns[6], we had to adapt our transformation process to deal with these *errors* as they occur quite often. This problem is less acute in the French UDs.

What should be a `lemma` is an hotly discussed subject on the UD GitHub, but there are still too many debatable lemmas such as *an*, *n't*, plural nouns etc. In one corpus, lowercasing has been applied to some proper nouns, but not all. We think it would be preferable to apply a more *agressive* lemmatization to lower the number of base forms to with help further NLP processing that is often dependent on the number of different types. The lexicons for jsRealB being sufficiently comprehensive for most current uses (34K lemmas for English

[5] `https://universaldependencies.org/u/overview/morphology.html`

[6] `https://universaldependencies.org/en/feat/Poss.html` indicates that `his` can be marked as a possessive pronoun.

and 53K lemmas for French), they are still unknown lemmas for specialized or informal contexts. But we found that, most often, *unknown* lemmas are symptoms of errors in the lemma or the part of speech fields that should be checked.

3.2 UD JSON to Syntactic Representation

The goal is to map the tree representation of the dependencies to a tree of constituents that can be used by jsRealB for realizing the sentence. According to the annotation guidelines, there are two main types of dependents: nominals and clauses which themselves can be simple or complex.

The head of a Syntactic Representation is determined by the terminal at the head of the dependencies. Dependencies are then scanned to determine if the sentence is negative, passive, progressive or interrogative depending on whether combinations of `aux`, `aux:pass` with proper auxiliaries (possibly modals) or interrogative `advmod` are found. When such a combination is found, then these relations are removed before processing the rest. The appropriate jsRealB sentence `typ` will be added to the resulting Universal Dependencies. For example, in Figure 1, the auxiliary `may` is removed from the tree and the sentence is marked to be realized using the *permission* modal.

All dependencies are transformed recursively; as each child is mapped to a SR, children list are mapped to a list of SR. Before combining the list of Syntactic Representations into a jsRealB constituent, the following special cases are taken into account, for English sentences:

1. a UD with a copula is most often rooted at the attribute (e.g. `mine` in Figure 3), it must be reorganized in order that the auxiliary is used as the root of a `VP`:

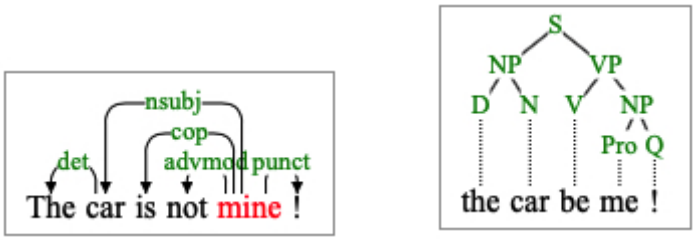


Figure 3: On the left, the dependency tree corresponding to the sentence The car is not mine!; at right, the dependency tree after transformation for which jsRealB realizes the same sentence.

2. A verb at the infinitive tense is annotated in UD as the preposition `to` before the verb, so this preposition is removed before processing the rest of the tree, it is reinserted at the end;

3. An adverb (from `advmod` relation) is removed from processing the rest and added to the resulting `VP` at the end;

4. If the head is either a noun, an adjective, a proper noun, a pronoun or a number, it is processed as a nominal clause mapped to a `NP` enclosing all its children UDs.

5. If the head is a verb: check if the auxiliary `will` is present, then a future tense option will be added to the verb; in the case of the `do` auxiliary, copy feature information (tense and person) into the jsRealB options.

6. Otherwise, bundle Syntactic Representations into a sentence `S`, the subject being the first child and the `VP` being the second child.

7. Coordinate `VP`s and `NP`s must also be dealt specially because the way that jsRealB expects the arguments of a `CP` is different from the way coordinates are encoded in UDs where the elements are joined by `conj` relations. in jsRealB, all these elements must wrapped in a global `CP`, the conjunction being indicated once at the start.

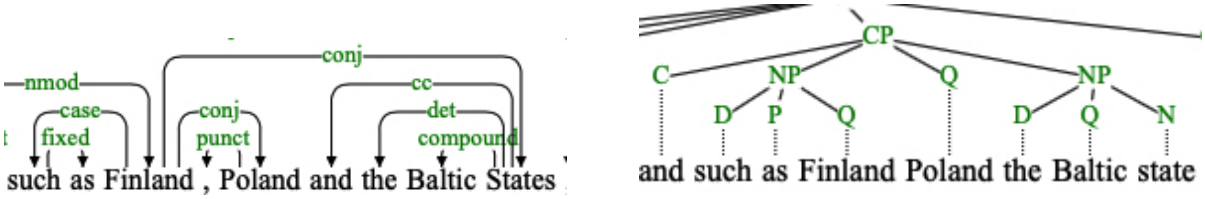


Figure 4: The graph at the left, a subgraph of the UD `w02013093` in `en_pud-ud-test.conllu`, illustrates the UD encoding of coordinated nouns Finland, Poland and the Baltic States; the right part shows the expected dependency tree by jsRealB.

We had originally implemented this tree to tree mechanism in Prolog (15 rules in 100 lines of commented and indented code) by reading the annotation guidelines and then refined by experimenting with the UD corpus. For UDRegenerator we *converted* this approach in JavaScript which unfortunately is much less appropriate for this type of transformation. On top of the fact that structure matching in JavaScript is more cumbersome than in Prolog, the feature that we missed the most and was more error-prone is the fact that in Prolog, it is easy to transform a tree to check for a certain condition and, when it is not met, backtracking resets it to its original state. This is not the case in JavaScript where tree modifications are much more delicate to *undo*, so we had to carefully find an ordering of transformations so that tree modifications to a certain step would not have adverse effects later.

This exercise in transforming UD structures to jsRealB revealed an important difference in their level of representation. By design UD stays at the level of the form in the sentence, while jsRealB works at the constituent level. For example, in UD, negation is indicated by annotating `not` and the auxiliary elsewhere in the sentence, while in jsRealB the negation is given as an *option* for the whole sentence. So as shown above, the structure is checked for the occurrence of `"not"` and an auxiliary to generate the `.typ({neg:true})` option for jsRealB (see Figure 3); these dependents are then removed for the rest of the processing. Similar checks must also be performed for passive constructs, modal verbs, progressive,

perfect and even future tense in order to *abstract* the UD annotations into the corresponding structure for jsRealB.

3.3 Working with French

As jsRealB can also be used for realizing sentences in French and that many UDs are available in French, we adapted for French the methodology described in the previous section. For morphology, we changed the lemmas for pronouns and numerals. Fortunately, the *ambiguity* between pronouns and determiners seldom occurs in the French UDs, so this step was more straightforward.

The transformation for clauses stays essentially the same as for English, except that there is no need to cater for the special cases for modals, future and infinitives.

4 Experiments

We experimented with version 2.7 of UD, the most recent at the time of writing. UDregenerator can be used interactively[7], but it can also be used as a node.js module to process a whole corpus and display at a console, the results and the differences between the original text and the regenerated one.

The following subsections describe our experience running UDregenerator on both the English and French corpora which shows that the system can handle all sentences and is quite fast: about 1,3 milliseconds per sentence on a commodity Mac laptop. As we use a symbolic approach, we do not distinguish between the training, development and test splits of a corpus, we consider them as different corpora. This allows an overall judgment on what we feel to be the precision of the informations in the UDs. The last subsection provides a more detailed analysis of a representative sample of the corpora.

4.1 English corpora

Table 1 shows statistics about the 13 English corpora that comprise 29 420 sentences of which 1 580 (5%) have non-projective dependencies. We did not use the three English ESL corpora because they do not provide any information about the lemma and the features of tokens, they only give their form and relation name.

We see that about 61% of the sentences are regenerated of which 53% are exact reproductions of the original, ignoring capitalization and spacing. The only exception being the `pronouns` corpus which uses a limited vocabulary and was manually designed to illustrate many variations of pronouns; in fact, we used it to design our pronoun transformations. Many of the differences are due to contractions (e.g. aint or he'll) for which jsRealB realizes the long form (is not or he will).

In order to limit the number of error messages, we decided to add a few `dubious` lemmas:

[7] `http://rali.iro.umontreal.ca/JSrealB/current/demos/UDregenerator/UDregenerator-en.html`

Corpus	type	#sent	#toks	#OK	#diff	#lerr	#t/s	%OK	%regen	%terr
ewt	dev	2002	25148	1359	585	911	12,6	68%	57%	4%
	test	2077	25096	1446	592	882	12,1	70%	59%	4%
	train	12543	204585	7669	3516	7150	16,3	61%	54%	3%
gum	dev	784	15598	447	209	519	19,9	57%	53%	3%
	test	890	15926	484	190	635	17,9	54%	61%	4%
	train	4287	81861	2410	1185	3024	19,1	56%	51%	4%
lines	dev	1032	19170	603	321	608	18,6	58%	47%	3%
	test	1035	17675	639	323	512	17,1	62%	49%	3%
	train	3176	57372	1928	1055	1711	18,1	61%	45%	3%
partut	dev	156	2722	73	31	122	17,4	47%	58%	4%
	test	153	3408	82	34	89	22,3	54%	59%	3%
pronouns	test	285	1695	270	133	15	5,9	95%	51%	1%
pud	test	1000	21176	501	245	774	21,2	50%	51%	4%
mean		2263	37802	1409	670	1304	16,8	61%	53%	3%
sample		60	1106	37	15	0	18,4	62%	59%	0%

Table 1: Statistics for the English UD corpora: for each corpus and type, it shows the numbers of sentences (#sent) and tokens (#toks); then the number of sentences that were reproduced by jsRealB without errors (#OK); the number of sentences that had at least one difference with the original (#diff); the number of tokens that had at least one lexical error (#lerr); the number of tokens per sentence (#t/s); percentages of sentences regenerated (%OK), of regenerated exactly (%regen) and of tokens in error (%terr). The next to last line displays the mean of these values over all corpora. The last line shows the statistics for the sample that is studied more closely in Section 4.3.

- best and better were added as lemmas, although we think that the appropriate lemma should be good specifying the `Degree` feature: superlative (`Sup`) or comparative (`Cmp`).
- & was added as a lemma for a conjunction, but it should be and.
- in formal English, adjective and nouns corresponding to nationalities start with a capital letter (e.g. American or European), but we also accept the lowercase form as lemma for these.

Corpus	type	#sent	#toks	#OK	#diff	#lerr	#t/s	%OK	%regen	%terr
fqb	test	2289	24135	1769	863	688	10,5	77%	51%	3%
gsd	dev	1476	35718	936	332	726	24,2	63%	65%	2%
	test	416	10019	259	114	213	24,1	62%	56%	2%
	train	14449	354662	9190	3106	7348	24,5	64%	66%	2%
partut	dev	107	1870	35	9	127	17,5	33%	74%	7%
	test	110	2603	34	13	112	23,7	31%	62%	4%
	train	803	24122	294	119	899	30,0	37%	60%	4%
pud	test	1000	24734	530	180	672	24,7	53%	66%	3%
sequoia	dev	412	10002	218	63	389	24,3	53%	71%	4%
	test	456	10048	269	79	337	22,0	59%	71%	3%
	train	2231	50517	1284	389	1737	22,6	58%	70%	3%
spoken	dev	909	10062	662	587	329	11,1	73%	11%	3%
	test	730	9991	497	466	322	13,7	68%	6%	3%
	train	1167	15172	818	726	521	13,0	70%	11%	3%
Mean		1897	41690	1200	503	1030	20,4	57%	53%	3%
Sample		60	1227	31	16	46	20,5	52%	48%	4%

Table 2: Statistics for the French UD corpora: for each corpus and type, it shows the numbers of sentences (#sent) and tokens (#toks); then the number of sentences that were reproduced by jsRealB without errors (#OK); the number of sentences that had at least one difference with the original (#diff); the number of tokens that had at least one lexical error (#lerr); the number of tokens per sentence (#t/s); percentages of sentences regenerated (%OK), of regenerated exactly (%regen) and of tokens in error (%terr). The next to last line displays the mean of these values over all corpora. The last line shows the statistics for the sample that is studied more closely in Section 4.3.

4.2 French corpora

The 14 French UD corpora provide 26 555 sentences of which 1113 (4 %) have non-projective dependencies. Ignoring the `partut` corpora, UDRegenerator regenerates about 64% of the sentences, which is slightly more than for the English corpora of which 53% are exact reproduction of the original text.

Given that the rate of correct regeneration for the `partut` corpora is around 33%, we examined them more carefully and saw that the lemmatization process for these corpora was often incomplete or erroneous. Here are a few examples taken from `fr_partut-ud-train.conllu`:

bad part of speech : `certain` (32 times) is determiner instead of an adjective; `comme` (18 times) is preposition instead of conjunction;

orthographic error : `region` (13 times) instead of `région`, `pubblicitaire` (5 times) instead of `publicitaire`;

bad lemma : `normes` (11 times) or `ressources` (8 times) whose lemma should be singular.

This is a good illustration of how UDREGENERATOR can help improve UD information.

In both French and English corpora, we found a few instances of bad head links for which regeneration produces words in the wrong order. The tree representation is particularly useful for checking these as there are crossing arcs. We noticed that most often this occurs in non projective dependencies, this is why the system flags these in order so that they identified more easily and checked.

4.3 Sample corpora

On top of the general remarks given above, we performed a more detailed study of a sample of 10 sentences from each of the 6 English and French test corpora for which we used UDREGENERATOR to recreate exactly the original sentence[8]. The statistics on the last line of Tables 1 and 2 show that these samples have roughly the same characteristics as the whole corpus from which they were taken. This experiment shows that JSREALB has an almost complete coverage of English and French grammatical phenomena, except for some specialized terminology which can be easily added to the lexicon or given as quoted words that will appear verbatim in the output. This was very seldom done in our case.

Once the lexical errors and missing words were corrected, 40 sentences from the 60 in the English sample could reproduced automatically. 12 had errors in their features, most often a missing person or number. 8 had part of speech errors, e.g. conjunction instead of adverb, a noun instead of a proper noun or an adjective instead of a gerundive verb. there were 7 cases of erroneous lemmas such as weaker instead of weak or choking instead of choke, We found two cases of erroneous head linking. Of course, those are very small numbers computed over only 60 sentences, the whole corpus being 490 times greater.

We also experimented with 60 sentences sampled from French corpora with the following results: 36 were regenerated automatically. 14 had errors in their features, often a bad value for Person or an error in Gender. 8 had errors in their lemma such as the plural form given as lemma and also bad encoding of elision (e.g. du must in some cases but be split to de le). 4 errors in parts of speech such as prepositions that should be adverbs. Of course this is only a very small sample (0,23%) of the whole corpus, but we think that is shows that there is a need to recheck the information in UD as it is often used as gold standard and sometimes even used as a *mapping* source for other lower-resourced languages.

5 Conclusion

This work which was first motivated for exercising JSREALB in order to measure its coverage, finally made us realize that UDs while being a source of useful linguistic information, should perhaps be checked by trying to regenerate the sentences from the given information. We are not aware any previous attempt to do such an experiment.

[8]These sample corpora including the equivalent JSREALB expressions are available at http://rali.iro.umontreal.ca/JSrealB/current/demos/UDregenerator/UD-2.7/

Of course, sentence regeneration is not foolproof because different feature combinations can produce the same sentences, but in many cases it helps to pinpoint discrepancies between what is specified and the expected outcome a process similar to the Schema validation of XML files. UDREGENERATOR is far from perfect, but it can surely be useful tool for doing some sanity checking on the lemma, part of speech, features and head fields. We hope that this work will help improve the precision of the wealth of useful information contained in UDs.

References

[1] Albert Gatt and Ehud Reiter. SimpleNLG: A realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation (ENLG 2009)*, pages 90–93, Athens, Greece, March 2009. Association for Computational Linguistics.

[2] Sylvain Kahane, Alexis Nasr, and Owen Rambow. Pseudo-projectivity, a polynomially parsable non-projective dependency grammar. In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics, Volume 1*, pages 646–652, Montreal, Quebec, Canada, August 1998. Association for Computational Linguistics.

[3] Guy Lapalme. Realizing Universal Dependencies Structures. Internal report, `http://rali.iro.umontreal.ca/rali/sites/default/files/publis/UDSurfR.pdf`, RALI-DIRO, 10/2019 2019.

[4] Simon Mille, Anja Belz, Bernd Bohnet, Yvette Graham, and Leo Wanner. The Second Multilingual Surface Realisation Shared Task (SR'19): Overview and Evaluation Results. In *Proceedings of the 2nd Workshop on Multilingual Surface Realisation (MSR), 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Hong Kong, China, 2019.

[5] Paul Molins and Guy Lapalme. JSrealB: A bilingual text realizer for web programming. In *Proceedings of the 15th European Workshop on Natural Language Generation (ENLG)*, pages 109–111, Brighton, UK, September 2015. Association for Computational Linguistics.

[6] Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajič, Christopher D. Manning, Ryan McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. Universal dependencies v1: A multilingual treebank collection. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016)*, pages 1659–1666, Portorož, Slovenia, May 2016. European Language Resources Association (ELRA).

[7] Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. Stanza: A python natural language processing toolkit for many human languages. In *ACL-2020 : System Demonstrations*, 2020.

[8] Fei Xia and Martha Palmer. Converting dependency structures to phrase structures. In *Proceedings of the First International Conference on Human Language Technology Research*, 2001.

[9] Daniel Zeman, Joakim Nivre, , and *many others*. Universal dependencies 2.7, 2020. LINDAT/CLARIAH-CZ digital library at the Institute of Formal and Applied Linguistics (ÚFAL), Faculty of Mathematics and Physics, Charles University.