

The JSREALB Text Realizer: Organization and Use Cases

Guy Lapalme
RALI-DIRO
lapalme@iro.umontreal.ca

January 14, 2021

Abstract

This paper describes the design principles behind JSREALB, a surface realizer written in *JavaScript* for English or French sentences from a specification inspired by the *constituent syntax* formalism. It can be used either within a web page or as a `node.js` module. We show that the seemingly simple process of text realization involves many interesting implementation challenges in order to take into account the specifics of each language. JSREALB has a large coverage of English and French and has been used to develop realistic data-to-text applications and to reproduce existing literary texts and sentences with Universal Dependency annotations. Its source code and that of its applications are available on GitHub.

1 Introduction

A text realizer sits at the very end of a text generation pipeline. Important decisions about *What to say* must have already been made, sentence structure and most content word choices must also have been decided. Final realization is an often *neglected* part in NLG systems because it is dubbed to be *pedestrian*, often associated with *glorified* format statements, even though its output is the only thing that the end user sees and that is used to evaluate the whole system. How reasonable is an output if word agreements are not properly made or if it consists of a mere list of tokens? This might be sufficient for machine evaluation, but it cannot practically be used in a production setting. A well-formatted and grammatically correct output is important for the *social acceptability* of a system. This is why we decided to look closely at this process.

A text realizer has much interesting work to do: it must take care of many language peculiarities such as number and person agreements, conjugation, word order and elision. On top of these tasks, JSREALB also allows creating many variations (e.g., negative, passive or interrogative) from a single affirmative sentence structure. We first briefly present JSREALB and compare it with existing text realizers.

2 jsRealB

JSREALB (JavaScript Realizer Bilingual) [6, 16] is a bilingual French and English text realizer that generates well-formed expressions and sentences and that can format them in HTML to be displayed in a browser (see the top of Figure 1 for an example). As its name indicates, JSREALB is written in JavaScript, a programming language that, when used in a web page, runs in the client browser. A web programmer who wishes to use JSREALB to produce flexible French or English textual output needs only add one line in the header of the page, similarly to what is done for other browser frameworks such as JQUERY. JSREALB is aimed at web developers. It carries out crucial tasks, from taking care of morphology, subject-verb agreement and conjugation, to creating entire HTML documents. JSREALB has been used for creating data-to-text systems. It has also been used as an intermediary for realizing sentences produced by a Prolog program taking input from AMR structures [10] or from Universal Dependencies in the context of the Surface Realization Shared Task (SR'19) at EMNLP [9] and for the WebNLG Challenge [3].

JSREALB was strongly influenced by SIMPLENLG [8]. In both cases, realization is achieved by programming language instructions that 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 a string. Like SIMPLENLG, JSREALB has the following components :

- a lexicon defining word category, gender, number, irregularities and other features;
- a list of morphological rules to determine the appropriate inflections, such as plurals and conjugations;
- an implementation of syntactic rules to properly order words in a sentence, perform agreement between constituents and carry out other interactions.

JSREALB integrates other useful tools, such as the spelling out of numbers and the wording of temporal expressions. Since it produces web content, it can add HTML tags for links, for formatting, to create headers and lists. It takes them into account so that they do not interfere with the proper processing of words within a sentence.

JSREALB has been under development in our lab since 2013.¹ It has been recently redesigned while keeping the same expression syntax. This paper is the result of this recent work, which revealed many intricacies that must be dealt with and that we wanted to document more formally.

2.1 Input to the realizer

In JSREALB, JavaScript expressions create data structures corresponding to the constituents of the sentence to produce. When the need arises to produce a string realization (i.e. its

¹The source code is now available on the [RALI-GitHub](#) with a tutorial and many demonstration applications.

`toString()` function is called), the data structure (a tree) is traversed to produce the tokens of the sentence, taking care of capitalization, elision and appropriate spacing around punctuation. It is also possible to wrap portions of text in HTML tags.

The data structure is built by function calls whose names² were chosen to be similar to the symbols typically used for constituent syntax trees³:

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

Features added to the structures using the dot notation can modify their properties. For terminals, their person, gender or number can be specified (e.g. `.pe(3).g("f").n("s")` for third person feminine singular). 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.

Since JSREALB expressions are standard JavaScript expressions built by functions and possibly modified with methods, no additional parsing of the input to JSREALB is necessary. Usual browser-based JavaScript development tools can thus be used to develop the realizer functions while allowing a seamless integration in webpages. The special terminal Q allows the insertion of *canned text*, a feature that caters for special needs and is often useful in practical applications, but is not discussed further here.

JSREALB can also be used as a standalone `node.js` module taking input created by an external system. From experience, we know that creating syntactically valid JavaScript expressions programmatically can be cumbersome and tricky. Therefore JSREALB also allows a JSON input format⁴ for which convenient APIs are available in almost all programming languages. This idea is similar in principle to the XML input format allowed in SIMPLNLG. The top part of Figure 1 shows the JavaScript notation for the realization of the utterance `He eats apples`. in which the word `apple` would appear emphasized when displayed in a browser.

3 Previous text realizers

Realizers have been an integral part of Natural Language Generation (NLG) systems since the eighties. Realizers such as KPML [1], SURGE [7], REALPRO [13], Forge [15] and GenDR [12] are based on linguistic theories, dealing with many details in the construction of sentences, which allows powerful realizations. However, that complexity hinders their ease of use: writing specifications for them requires an intimate knowledge of the underlying theory.

²Traditionally 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.

³See the [documentation](#) for the complete list of functions and parameter types.

⁴The specifics are described in [this document](#)

```

S(
    Pro("him").c("nom"), // Pronoun (citation form), nominative case
    VP(V("eat"),        // Verb at present tense by default
        NP(D("a"),      // Noun Phrase, Determiner
            N("apple").n("p") // Noun plural
        ).tag("em")      // add an <em> tag around the NP
    )
)



---


Lexicon lexicon = new XMLLexicon(); // default simplenlg lexicon
NLGFactory nlgFactory = new NLGFactory(lexicon);

NPPhraseSpec np= nlgFactory.createNounPhrase("a", "apple");// create NP
np.setFeature(Feature.NUMBER,NumberAgreement.PLURAL); // set plural
SPhraseSpec s = nlgFactory.createClause("he","eat", np);// create sentence
DocumentElement sentence = nlgFactory.createSentence(s);

Realiser realiser = new Realiser(lexicon);
NLGElement realised = realiser.realise(sentence);
System.out.println(realised.getRealisation());



---


from CoreNLG.DocumentConstructors import TextClass
class Content(TextClass):
    def __init__(self, section):
        super().__init__(section)
        self.text = self.free_text(
            "he",
            "eats",
            self.nlg_tags("em",text="apples")
        )

```

Figure 1: Top: JavaScript functional with comments at the right realized by JSREALB as: *He eats apples*. Middle: SIMLENLG Java statements to print a roughly equivalent sentence: *He eats some apples*. Bottom: Python class from CoreNLG project to create a web page displaying *He eats apples*.

In fact, most existing realizers are considered so complex that SIMPLENLG [8], as its name implies, defines itself by its ease of learning and of use. Words, phrases and other structures are Java objects created and manipulated intuitively by a programmer and easily integrated into a Java project. While its principles somewhat limit the power of its realizations compared to other systems, these realizations are largely sufficient for many uses. SIMPLENLG has been used in a variety of text generation projects.⁵ Our realizer was originally inspired by SIMPLENLG for which we have developed a bilingual (French and English) version [18].⁶

But with the rise of web applications, we found that the use of Java for building a text realizer was complicated by the need to create a distinct web server for building new web pages and creating an XML structure for realization. As the Java notation is relatively verbose, we decided to develop a notation that mimics the usual notation of the constituent grammar. Embedding the realizer in the web page itself greatly simplifies the architecture of the application. However JSREALB would probably never have been created without our previous experience with SIMPLENLG. The middle part of Figure 1 shows the input to create an almost equivalent sentence to the one written in JavaScript. SIMPLENLG does provide some HTML formatting but only at the paragraph level (list and title) and not at the token level; the plural of the indefinite article *a* is realized as *some*.

Many data-to-text information systems can *get away* with template-based text generators in which it is sufficient to be able to deal with the description of list of items. Van Deemter et al. [17] argue that these types of systems are interesting and useful and not necessarily *inferior* to the more general approaches.

One the first template system to be described in detail is YAG [14] which bridges the gap between linguistic theories and the need for practical data-to-text text generation systems.

ROSAENLG⁷ is a JavaScript NLG library based on the Pug templating engine⁸. It supports languages such as English, French, German, Italian and Spanish with some features such as agreement within noun phrases, a few tenses and agreement at the third person of some simple tenses. ROSAENLG was developed for realizing some simple data to text applications and is especially tuned for outputting lists of objects and properties using appropriate commas and a conjunction at end of the list. ROSAENLG being built over a template engine, it is not so well integrated within a web page as JSREALB because a distinct specific file must be written and transformed into JavaScript before being integrated in the web page. Moreover the linguistic coverage of JSREALB, for French and English at least, is much more complete than what is offered by ROSAENLG.

CORENLG⁹ is a Python library for Natural Language Generation providing some tools to structure and write NLG projects, agreement between elements must be based on external resources. Like JSREALB, it deals with some typographical conventions and elisions and also allows choosing randomly between synonyms, but it is mostly grammar unaware. A

⁵SIMPLENLG is freely available at <https://github.com/simplenlg/simplenlg>

⁶Available at <https://github.com/rali-udem/SimpleNLG-EnFr>

⁷<https://rosaelng.org>

⁸<https://pugjs.org>

⁹<https://github.com/societe-generale/core-nlg>

simplistic example of its formalism is shown at the bottom part of Figure 1.

JSREALB differs from COMPROMISE¹⁰ which is a low-level JavaScript API for both parsing English and extracting information. It dubs itself as *trying its best, and being small, quick, and usually good-enough*. It also provides a few functions for dealing with some English constructions: conjugation at simple tenses and plural of nouns. COMPROMISE stays at level of the JavaScript program and generates list of tokens, while JSREALB is oriented toward the specification of the constituent structure from which the basic tools are automatically called.

Text realizers are important in commercial systems such as ARRIA¹¹ that provide sophisticated templates that can be customized with a specialized programming language within well-developed web-based environments. The strength of the system lies with the ease of linking to different types of data sources and analytic tools, but it requires learning a specialized language instead of relying on the underlying and well-known JavaScript formalism.

Neural models for end-to-end data-to-text generation have been proposed, but they focus on the whole process and pay relatively little attention to the final realization step, which we address here. Castro Ferreira *et al.* [4] found that having intermediate steps in the generation process generally leads to better texts than end-to-end systems. This is one of the reasons why we devoted time to study this *final* process of NLG.

We now detail some interesting aspects of the organization of JSREALB to take care of the automatic agreement between constituents which is an important feature in order to limit the number of annotations to give and to add some flexibility when individual components are combined in different ways.

4 Steps of the realization process

4.1 Structure creation

To illustrate the steps to go from a JSREALB expression to the corresponding English sentence, we start with the sentence structure in Figure 1 whose corresponding internal data structure is shown in Figure 2.

Diagrams in this paper use these graphical conventions:

- a **Phrase** is shown as a rectangle containing its name on the first line and possibly options on the following line;
- a **Terminal** (to be realized as a string of words, possibly none) is displayed as a rounded rectangle containing its name followed by its *lemma* in parentheses; options may appear on the second line. On a third line, will be the realization string once it has been determined.

¹⁰<http://compromise.cool>

¹¹<https://www.arria.com/>

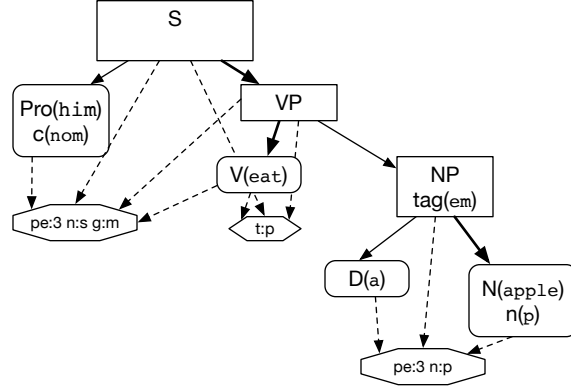


Figure 2: Internal data structure corresponding to the expression in Figure 1

- Full line arrows indicate parent-child relationships, lines with larger width link a parent to its *head* child. Dashed lines link a constituent with its *shared* properties. Shared properties are first associated with terminals: they are person (**pe**), number (**n**) and gender (**g**) (shown as a flat octagon in Figure 2) for nouns, pronouns, adjectives, determiners and verbs; verbs also have two other shared properties, tense (**t**) and auxiliary (**aux**) (in French only), shown as a flat hexagon in Figure 2.

These properties are *shared* to ensure that proper agreement is made between constituents¹² so that when a property of constituent is changed then all dependent constituents will be changed accordingly.

When phrases are created from terminals and other phrases, the shared properties of the parent are set to its head's so that modifications to the parent will be propagated to its children. In Figure 2, the tense of the VP is set to that of the V, and the number of the NP is set to that of the N. But other links must also be added: a VP must also be linked to its subject in order to properly agree in person, gender and number, this is why the VP is linked to the shared property of the Pro. Agreement must also be made between children of a NP: a determiner, adjectives must agree with the head of a NP, so they all share the same properties.

Some properties (such as formatting) are not shared, they only apply to the current constituent. The only shared properties are those dealing with agreement between constituents.

We now explain how to go from an input structure to an English sentence. To realize a sentence, the `toString()` function performs two tasks:

- *stringification* which computes the realization property of a terminal possibly modified with some formatting;
- *detokenization* which creates a single final string for a phrase.

The next sections detail these steps.

¹²In Prolog, this would be achieved by unification between shared variables that are changed as soon as one of its occurrences is changed.

4.2 Stringification of constituents

The production of strings from terminals and phrases is carried out recursively. Realization strings are produced for each terminal, but also for each phrase at each level of the tree. This process builds lists containing the original terminals, using information from the original data structure (e.g., number, gender or agreement links) for proper conjugation and declension.



Figure 3: Tokens from Figure 2 once their `realization` field is filled in.

4.2.1 Computing the realization property

The `toString()` function of a *Phrase* calls `toString()` on its children to build a list of the original terminals with their realization field filled in.

The `toString()` function of a *Terminal* applies declension or conjugation rules taking into account the gender, number, person and possibly tense information in order to fill in the `realization` field of the terminal. Figure 3 illustrates a few interesting cases.

- `Pro("him").c("nom")` is realized as **he** because the declension rules set its gender to masculine, its person to third and its number to singular;
- `VP("eat")` is realized by the string **eats** because the verb is conjugated at the present tense by default as given by its shared property and, as shown in Figure 2 it is as third person singular, from the shared property of its subject, the `Pro`;
- `D("a")` produces an empty string because it is a plural indefinite article as given by its shared property with the `N`.
- `N("apple").n("p")` produces the HTML snippet `apples`, the plural declension of the lemma **apple**. As will be explained in the next section, the formatting `em` tags come from the surrounding NP.

Using the shared properties allows global modifications to the original expression although it does not override locally set properties. For example, adding `.t("ps")` at the end of the expression in Figure 1 indicates that the whole expression should be at the simple past. The modified expression would then be realized as **He ate** `apples`.

4.2.2 Formatting the realization property

Formatting is carried out in a series of optional steps:

1. Apply language-dependent token modification rules taking into account the surrounding words:

- English:
 - the determiner **a** which changes to **an** depending on the first letter of the next word, most often a vowel.
 - French:
 - *elision* for determiners such as **le**, **ce**, **la**, etc., or conjunctions such as **que**, **puisque**. If the next word starts with a vowel, then the current word's last letter is changed to an apostrophe ' , which will be joined with the next word. In fact, rules for French are a little more complicated because this case also applies to some words starting with an **h**.
 - *euphony* for adjectives such as **beau**, **fou** or **vieux** which are changed to **bel**, **fol** or **vieil** if the next word starts with a vowel.
 - *contraction* combining words such as **de le** to **du**, **si il** to **s'il**, **à les** to **aux**.
2. Modify the realization string to insert strings that should appear before `.b(...)`, after `.a(...)` or around `.ba(...)` the realization string;
 3. Surround the realization string with HTML tags given by `.tag(...)`.

In our example, only the HTML tag **em** was added when formatting the NP, Section 6.4 will discuss why formatting can interfere with the realization process.

4.3 Detokenization

At this stage, we have a list of tokens. The following step, *detokenization* is applied to produce a well-formed English sentence. In our running example, this produces **He eats apples**.

Detokenization creates a single string by inserting appropriate whitespace between the realization strings of the terminals. When detokenization is applied to the top-level **S**, it capitalizes the first letter of the sentence and adds a full stop at the end, unless indicated otherwise.

We thus see that even for our example's simple *three words* sentence, many steps are needed to produce a well-formed sentence.

5 Structure modifications

One very useful feature of JSREALB, inspired by a similar one in SIMPLENLG, is the fact that a single affirmative sentence structure can be realized as negative, passive, interrogative or with a modality verb by setting a *type* flag on the sentence or verb phrase structure without having to change its structure. More than 2000 variations are possible from a single English affirmative sentence¹³. This capability of the realizer proved very useful when

¹³See this [demonstration of these variations with example sentences](#)

verbalizing AMR structures [10] in which negation is only indicated by a negative polarity role on an otherwise affirmative structure. We could just translate the rest of the sentence as an affirmative one and add a negative flag at end. We also used this feature to create a training corpus of sentences showing different forms from a single affirmative sentence that had been automatically parsed.

But to realize these modifications, JSREALB must internally modify the original sentence structure by adding new words and changing word order while keeping agreement links intact. This section highlights a few interesting cases.

5.1 Negation

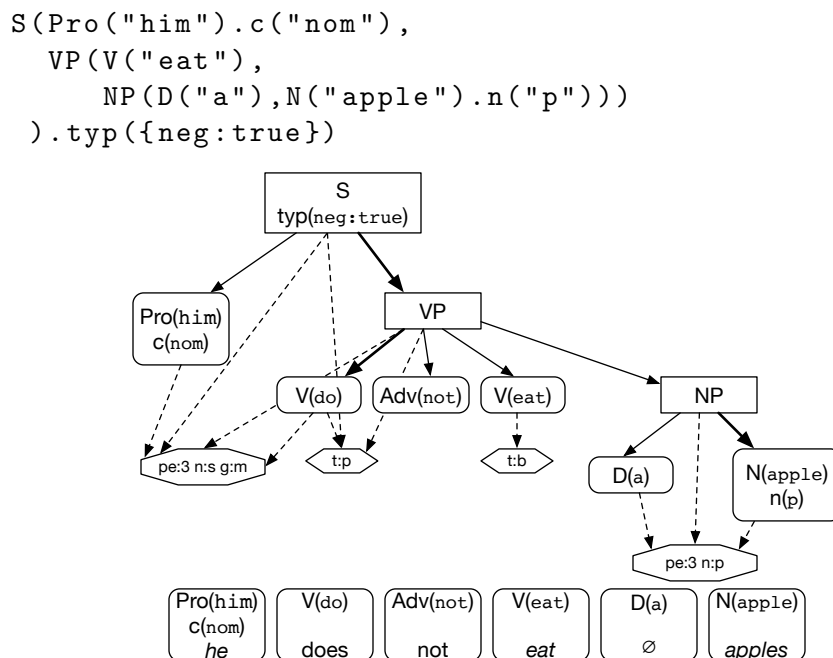


Figure 4: At the top is the example of Figure 2 with a negation flag set; in the middle is shown the modified data structure produced by JSREALB; the bottom shows the resulting list of tokens.

We now explain how a negative sentence is realized when `.typ({neg:true})` is added to the top-level `S` constructor of our previous example. To simplify the diagram, we removed the `.tag("em")` from the `NP`. Figure 4 shows the JavaScript notation (top) and the resulting data structure (middle) once the `.typ(..)` is applied. The structure of the `VP` is modified. The auxiliary `do` and the adverb `not` are inserted in front of the verb. The head of the `VP` is now the auxiliary `do` which shares its property with the `Pro`. The tense of the original verb is now infinitive (`b` for *base form*). The *stringification* and *detokenization* processes are the same as in the previous example. The end result is the string **He does not eat apples**.

5.2 Passivization

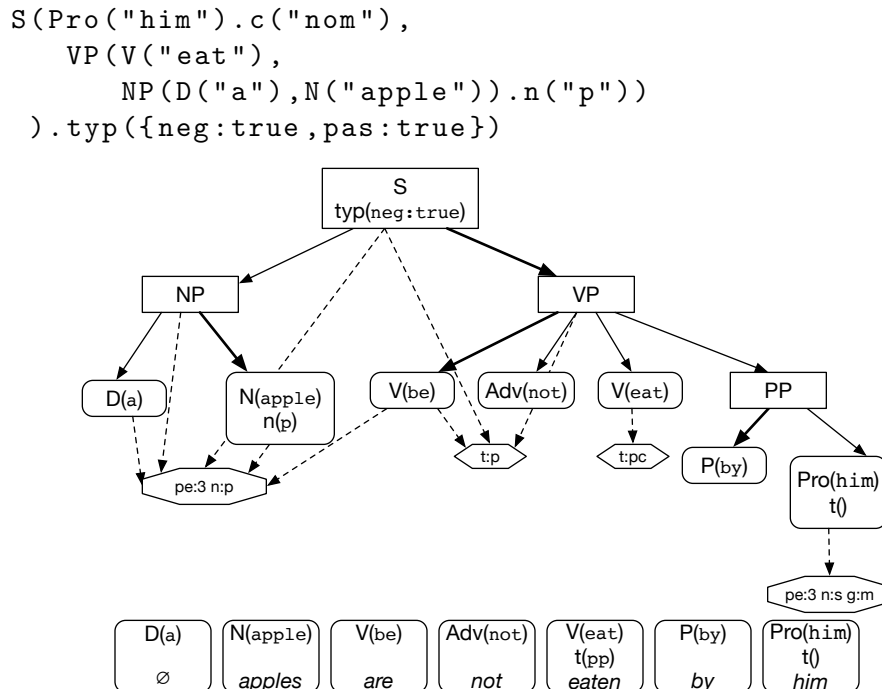


Figure 5: Figure 4 with passive flag set and the resulting data structure and tokens after passivization has been performed. The final string is **Apples are not eaten by him**.

Figure 5 shows the transformation to the *passive* form of our running example. It is now realized as **Apples are not eaten by him**. In the JavaScript notation at the top, `pas:true` is added in the `.typ` call. The middle part of the figure shows the data structure once these negative and passive transformation processes are applied: the object becomes the subject, the main verb becomes `be`, the original verb's tense is changed to the past participle, and the original subject becomes a prepositional phrase starting with `by`. In our example, the original subject being a nominative pronoun, it has to be changed to its tonic form. The head of the `VP` is now `be` whose shared properties for the subject is the `NP`. As the subject changes, the shared link for the `S` must also be changed.

5.3 Pronominalization

Another interesting structure transformation is the pronominalization process to change a noun or prepositional phrase to a pronoun having the same number and gender as the original phrase. That process is indicated by applying `.pro()` method the phrase. We use an example in French because it illustrates more interesting transformation processes coupled with agreement difficulties within the realization process.

```

S(Pro("lui").c("nom"),
  VP(V("donner").t("pc"),
    NP(D("un"),N("pomme"))).pro()
)

```

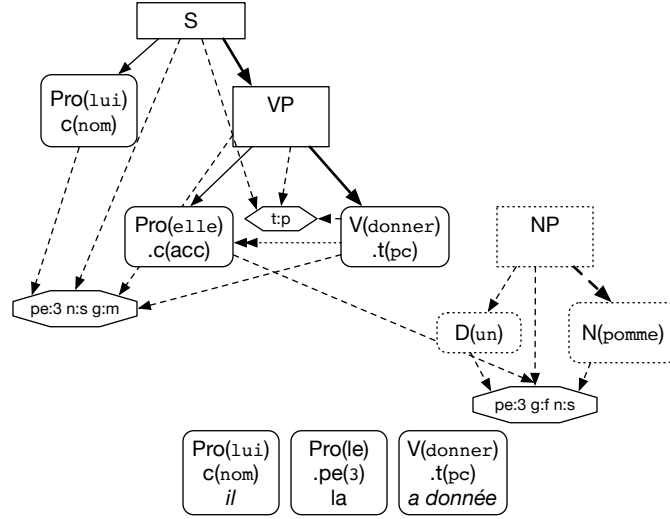


Figure 6: JavaScript specification of a French sentence and the resulting data structure after pronominalization

Not taking into account the call to `.pro()`, the top of Figure 6 shows a variation applied to the French version of our running example. It is realized as *Il a donné une pomme*. (*He has given an apple.*) in which the verb is conjugated to *passé composé*, corresponding approximately to the English *present perfect* in its form. When pronominalization is applied, the realized sentence is *Il l'a donnée*. (**He it has given.*)

Figure 6 shows the data structure once the call to `.pro()` has been processed: the noun used as the direct object must be replaced by an accusative pronoun. In this case `Pro("elle").c("acc")` agreeing in gender (feminine) and number (singular) with the original noun that will not be realized. This is why it is shown as a dashed rounded rectangle in the figure.

In French, some *interesting* peculiarities must be taken into account:

- a pronoun used as direct object must appear **before** the verb (except for an imperative verb), so the order of the children of the `VP` must be changed while linking the pronoun to the properties of the original noun to ensure proper gender and number;
- A verb conjugated at the *passé composé* is built using an auxiliary verb (**avoir** (*have*) in this case) followed by a past participle;
- a past participle with the **avoir** auxiliary must agree in gender and number with its direct object when it appears before in the sentence. This was not the case in the original sentence, but once the pronoun is shifted before the verb, the past participle

must agree with the pronoun. This is why in French a supplementary link (shown here as a dotted line with a double arrow head) between a verb and its direct object. So when the past participle is realized, it checks its position and then uses the gender and number of the direct object, here the **Pro(elle)** whose shared properties are the ones of the original NP. But the auxiliary of the main verb must agree with the original subject. Another case of past participle agreement occurs with the *être* (*be*) auxiliary must agree with its subject. We will see later an instance of this case.

The modified structure is detokenized as in the preceding section to produce the tokens shown at the bottom of Figure 6. Elision transforms *la a* to *l'a* and the formatting is realized as: *Il l'a donnée*.

5.3.1 Further modifications

French negation combined with pronominalization raise delicate word ordering problems. Table 1 illustrates modifications of the example of Figure 6 in which the differences between each example are underlined. The second column gives the JSREALB specification and the third column shows the corresponding French realization with an English transliteration in italics.

- In line 1, *ne ... pas* wraps around the verb and the preceding pronoun.
- Line 2 adds an indirect object using a PP (prepositional phrase) at the end of the sentence.
- When the PP is pronominalized (line 3), the pronoun must also appear before the verb; as the pronoun is combined with *à*, then the dative form *lui* must be used.
- Line 4 shows the passive form in which *par lui* (*by him*) comes from the tonic form of the pronoun that was the subject. In this case, the past participle is still singular feminine but for a different reason. Passivization takes the direct object and makes it the subject of the passive sentence used with the *be* auxiliary. So the past participle must agree with the new subject which, in this case, was the original object occurring before the verb.

These examples might seem contrived, but these cases appear frequently in real world texts and it is important that they be handled correctly to create correct French texts. They also show the importance of having a distinct realization step that can take care of these cases and make it possible to realize many variations from a single input structure.

English passivization also raises interesting problems especially when combined with perfect tenses, progressive mood and modal verbs (possibility, permission, necessity ...) for which *affix hopping* rules [5, pp 38–48] are implemented. It is also possible to modify an affirmative structure to get an interrogative form (Who? What? Yes or No ?...)

1	<pre> S(Pro("lui").c("nom"), VP(V("donner").t("pc"), NP(D("un"),N("pomme")).pro())).typ(neg:true) </pre>	<p>Il <u>ne</u> l'a <u>pas</u> donnée.</p> <p><i>*He it did <u>not</u> give.</i></p>
2	<pre> S(Pro("lui").c("nom"), VP(V("donner").t("pc"), NP(D("un"),N("pomme")).pro(), PP(P("à"),NP(D("le"),N("fille"))))).typ({neg:true}) </pre>	<p>Il ne l'a pas donnée <u>à la fille</u>.</p> <p><i>*He it did not give <u>to the girl</u>.</i></p>
3	<pre> S(Pro("lui").c("nom"), VP(V("donner").t("pc"), NP(D("un"),N("pomme")).pro(), PP(P("à"), NP(D("le"),N("fille")) .pro())).typ({neg:true}) </pre>	<p>Il ne la <u>lui</u> a pas donnée.</p> <p><i>*He <u>it</u> to her did not give.</i></p>
4	<pre> S(Pro("lui").c("nom"), VP(V("donner").t("pc"), NP(D("un"),N("pomme")).pro(), PP(P("à"), NP(D("le"),N("fille")) .pro())).typ({neg:true,pas:true}) </pre>	<p>Elle ne lui <u>a</u> pas <u>été donnée</u> par lui.</p> <p><i>*It <u>has</u> not <u>been given</u> <u>to her</u> by him.</i></p>

Table 1: JavaScript specifications and the corresponding realizations for combinations of pronominalization, negation and passivation as produced by JSREALB.

6 Other useful features to take into account

6.1 Incremental building of *phrases*

Although our previous examples of JSREALB expressions have been created manually, these expressions are most often built by programs by invoking the API's JavaScript functions. Moreover, it may happen that not all arguments to a phrase are known before calling the function. For example, the subject and the verb can be determined in one part of a program, but complements are only later specified. To account for this possibility, JSREALB allows adding new elements to an existing phrase. The `add(Constituent,position)` method inserts either a phrase or a terminal to the current phrase at a certain position given by a non-negative index. If `position` is not specified, the constituent is added at the end.

For example

```
S(Pro("him").c("nom"),
  VP(V("eat"),
    NP(D("a"),N("apple").n("p")).add(A("red"))))
).add(Adv("now").a(", "),0)
```

is realized as *Now, he eats red apples.* The adjective *red* is added at the end of the NP but, because almost all adjectives in English are placed before the noun, it appears before the noun. The adverb *now* followed by comma is inserted at the start of the sentence because the `position` is set to 0.

This dynamic feature explains why the realization process is launched at the very last moment (i.e., when a string is needed) and not as the structure is being built. For technical reasons `.add(...)` is not available in the JSON format, but this is not a big limitation because this dynamic phrase building should be performed within the host system before creating the JSON output.

6.2 Coordination

Data to text applications must often output lists of objects that can be realized using template based system like CoreNLG and RosaeNLG described in Section 3. But linguistically, these descriptions use coordinated phrases which are specified in JSREALB with a phrase CP in which a conjunction must be given with as many elements as needed. In the corresponding realization, all elements except for the last one are separated by a comma followed by the conjunction and the final element. For example:

```
S(CP(C("and"),NP(D("the"),N("apple")),
      NP(D("the"),N("orange")),
      NP(D("the"),N("banana"))),
  VP(V("be"),A("good")))
```

is realized as *The apple, the orange and the banana are good.* in which JSREALB takes into account that the subject is now plural because of the *and*. If `C("or")` had been given, the verb would have been realized as singular.

Such coordinated sentences are often built incrementally and, in some cases, only one element is required. For example,

```
S(CP(C("and"),NP(D("the"),N("apple"))),
  VP(V("be"),A("good")))
```

is realized as **The apple is good.** in which the conjunction is ignored and the number stays singular, unless of course the single subject is plural.

The generation of the tokens for a coordinated phrase must also be performed at the *last minute*, i.e., during stringification, during which this special case must be checked.

6.3 Reusing jsRealB expressions

One of the advantages of using a programming language for creating text is the fact that repetitive structures can be coded once and reused as often as needed. JSREALB expressions, being JavaScript objects, can be saved in variables, received as parameters or returned as result by functions. This fact is heavily used in the [demos and tutorial](#).

As we have shown earlier, options may modify the original structure of the expression. It is worth pointing out that reusing a modified expression will thus realize the modified object and not the original one according to the usual JavaScript behavior. For example, given the assignment:

```
let apple = NP(D("a"),N("apple"))
```

the expression

```
S(Pro("him").c("nom"),
  CP(C("and"),
    VP(V("eat"),apple),
    VP(V("love"),apple.pro()))))
```

is realized as **He eats an apple and loves it** which is expected. But later in the program, the expression

```
S(apple,VP(V("be"),A("red")))
```

will be realized as **It is red.**, in which the pronominalization of **apple** is still in effect.

If this is not what was intended, then a new **apple** object must be created before pronominalization. To achieve this, one can call `clone()` which creates a new copy of the data structure. This is implemented by traversing the object and creating a string that corresponds to the JSREALB expression for building this object. The resulting string is then evaluated in the current context to build a copy of the original expression.

So our previous **S** could have been coded as

```
S(Pro("him").c("nom"),
  CP(C("and"),
    VP(V("eat"),apple),
    VP(V("love"),apple.clone().pro()))))
```

after which


```
S(apple, VP(V("be"), A("red")))
```

is realized as **The apple is red.**

Seasoned JavaScript programmers might prefer defining arrow functions instead of variables that will create a new data structure at each call. For example, we might define

```
let apple = () => NP(D("a"), N("apple"))
```

and call `apple()` when needed. This also allows parametrizing the expression at each call.

6.4 Formatting

JSREALB being aimed at web developers, it provides the generation of HTML tags using the method `tag(name, attributes)` where `attributes` is an optional object whose keys are attribute names and values are the corresponding attribute values. When this method is encountered, it saves the parameters and the values in the constituent. At *stringification* time, these values are used to create the final string while taking into account HTML tags. For example

```
S(Pro("him").c("nom"),  
  VP(V("eat"),  
    NP(D("a"),  
      N("apple")  
        .tag("a",  
            {href: 'https://en.wikipedia.org/wiki/Apple'})))  
  ))
```

is realized as

He eats an `apple`.

in which the elision is performed between **a** and **apple** even though in the realized string the first letter of **apple** does not appear immediately after **a**, but it will be the case when it is displayed.

The punctuation before, after and around constituents is dealt similarly. The appropriate values of the strings to be inserted are saved within the constituent structure and used during the *stringification* process.

Note that this implementation choice implies a *small* limitation: HTML and other formatting can only be performed at constituent boundaries.

6.5 Other issues

JSREALB having been designed for use in the context of data-to-text systems, it must deal with the proper generation of numbers. On top of the usual formatting, it also deals with the writing of numbers in letters while taking into account the number agreement for the noun:

`NP(NO(1).dOpt({nat:true}),N("plane"))` realized as one plane while `NP(NO(3).dOpt({nat:true}),N("plane"))` gives three planes)

Number agreement for the noun phrase depends on the value of the number, note that rules for French and English slightly differ in this respect. It is also important to format dates in different forms: in letters or numbers, perhaps ignoring some components (e.g. show only date or time). Relative dates (e.g. **tomorrow**, **yesterday**, **last Monday**,...) can also be realized.

JSREALB also allows choosing randomly between a list of alternatives. This feature is useful for varying equivalent wordings for similar informations. It is used in a demonstration program for explaining subway routes between Metro stations of Montréal described in the next section. It could also be used to create random text generators in the spirit of Perchance.¹⁴

7 Applications

The [JSREALB GitHub demo repository](#) shows many examples of use of JSREALB for some specific features: conjugation of any verb and declension of any word, showing all possible sentence modifications (e.g. negation, passivation, interrogation, etc.) from a single sentence structure, building a sentence structure using menus or even create a random sentence.

JSREALB has also been used to create variations on existing texts such as *Les exercices de style* of [Raymond Queneau](#), also in its English version. It was also integrated in a web page for illustrating the flowchart for *L'augmentation* by [Georges Perec](#). The fact that it has been used to reproduce verbatim one version of the classic story of *Little Riding Hood* in both French and English illustrates that the coverage of JSREALB is quite extensive as it will be discussed in the next section.

JSREALB has also been used for Data to Text applications such as the [E2E challenge](#) although JSREALB was developed after the competition and thus too late to be part of the competition and be evaluated as such. There is also a demonstration of the description of a list of events; in fact this application was one of the first application developed with JSREALB and served as an initial motivation.

Two applications show text realization from data after non-trivial computations in both French and English (see Figure 7)

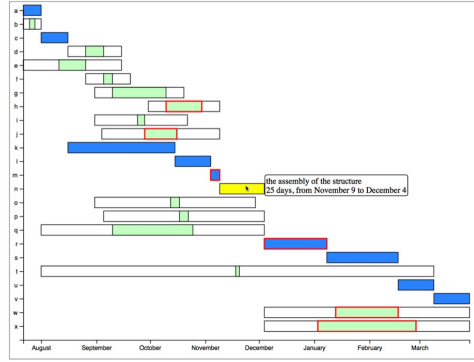
- The description of the construction of a house given information about tasks, the duration and the precedence relations between them. The system first computes the critical path to find the start and end times of each task. It then creates a graphic for displaying the PERT diagram and an accompanying text to explain the steps to follow. It is possible to interactively change the start date and to explore the graphic with the mouse which also uses JSREALB to generate the text of the tooltips.
- The description of an itinerary in the Montréal Métro network. The system shows an interactive map of the Montréal Métro station. When a user clicks two stations,

¹⁴<https://perchance.org/welcome>

Input data

The table on the left show the input data: for each task are specified: an id, its length in days and tasks that depend on it. The starting date of the project can be changed here: 2018-07-22

Id	Description	# days/Precedes
a	the study of the heating	10/c,q,t
b	the study of the roof	3/c,u
c	the study of the frame	15/k,d,p
d	the study of the reinforced concrete	10/f,o
e	the preparation of the land	15/g,i,o,f
f	the installation of the land	5/g,i,o
g	the earthwork for the shafts	30/h
h	the concrete for the shafts	20/n
i	the ground for the stringers	4/j
j	the concrete for the stringers	18/n
k	the construction of the frame in the factory	60/l,m
l	a first coat of paint in the factory	20/m,s
m	the move of the frame	5/n
n	the assembly of the structure	25/x,w,z
o	the earthwork for the canalizations	5/p
p	the masonry for the canalizations	5/w,z
q	the supplies for the heating	45/r
r	the installation of the heating	35/s
s	other coats of paint	40/u
t	the study of the electricity	2/v
u	the isolation of the roof	20/v
v	the installation of the electricity	20
w	the paving	35
x	the sealers	55



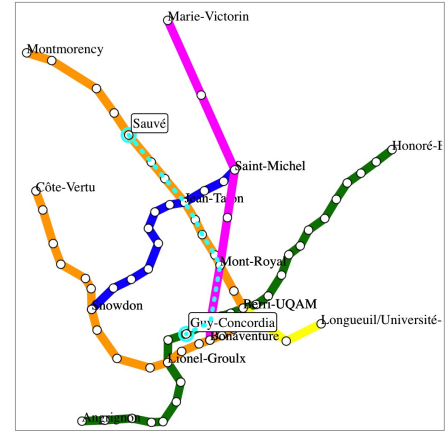
The system computes the critical path and the earliest and latest dates to which the tasks can take place so that the final project is not delayed. It produces then a text (realized by jsrealb). Hovering the mouse on a task in the graphic highlights it in yellow and displays a tooltip showing informations about this task which is displayed in blue in the generated text and in the table. The immediately preceding and following task are displayed with a red border.

Text describing an optimal sequence of tasks starting from July 22, 2018

To end the construction of the building, at least two hundred and fifty days will be needed. You should start plan the heating, plan the roof and prepare the land.

On August 1, 2018 you should prepare to plan the frame, to supply for the heating and to plan the installation of the electricity. On August 16, 2018 you should prepare to plan the study of the reinforced concrete and to build the frame in the factory. On August 26, 2018 you should prepare to install the land. On August 31, 2018 you should prepare to do the earthwork for the shafts, to ground the stringers and to do the canalizations. On September 4, 2018 will be the start of the concrete for the stringers. On September 5, 2018 will be the start of the masonry for the canalizations. After you will do the concrete for the shafts. On October 15, 2018 will be the start of a first coat of paint in the factory. Then you should move the frame. On November 9, 2018 will be the start of the assembly of the structure. On December 4, 2018 will be the start of the installation of the heating, the paving and the sealers. After you will do other coats of paint. On February 17, 2019 will be the start of the isolation of the roof.

You will finish by the installation of the electricity. The construction of the building will be completed on March 29, 2019.



Go from Sauvé to Guy-Concordia

You are 37 minutes from your destination, board on the orange line towards Côte-Vertu up to Mont-Royal

Switch on the pink line towards Bonaventure for only one stop.

At last, you will find Guy-Concordia, second station on the green line towards Angrignon

Figure 7: Screen shots of web based application with texts realized using JSREALB. On the left are described the steps for building a house after critical path computation, on the right is an interactive map of the Montréal metro which shows the itinerary between two stations.

the systems realizes a text describing the itinerary to go from the first station to the second after having computed the shortest path between them.

JSREALB has been embedded into an [Observable notebook](#) in which it is possible to change the existing expressions and see how they are realized.

JSREALB can also be used from a *console* as a NODE.JS module. An interactive IDE allows the testing of expressions, the consultation of lexicon and also *lemmatization*, i.e. given a word, show all possible JSREALB expressions to realize it. For example, given `springs`, it returns `N("spring").n("p")` and `V("spring")`.

This NODE.JS module can be used as a web server to realize sentences produced by a system written in another programming language. We have used it from a system written in Prolog and another in Python for developing symbolic realizers the context of recent NLG competitions: Surface Realization Shared Task 2019 [9] and Web NLG challenge 2020 [3].

8 Coverage

8.1 Orthography

JSREALB can realize bilingual texts mixing French and English words and syntactic constructions. Each word or phrase is associated with a specific language. It comes bundled

with two lexicons (3 200 entries for French and 5 200 for English) that cater for most frequent uses: it has declension tables for all determiners, nouns, adjectives and pronouns in their tonic and clitic forms, it has conjugation tables for all English and French verbs even defective ones. These are used by JSREALB itself for localizing the warning messages, in the appropriate language, when erroneous input is detected such as bad values for parameters or properties not applicable to a given constituent.

It is easy to add new words to the lexicon by linking to the appropriate inflection table. The IDE provides a way for finding the right table, for example by providing entries with a given ending. For more sophisticated applications, the JSREALB [data repository](#) provides two larger lexicons, 52 522 words for French and 33 932 for English, already linked to the conjugation and declension tables.

On top of all simple tenses for French and English verbs for all persons and numbers, JSREALB can also realize verbs with auxiliaries such as *passé composé* or *subjonctif plus-que-parfait* in French or progressive or perfect verbs in English. It can also realize comparative and superlative forms of adjectives.

8.2 Syntax

JSREALB has all the necessary syntactic constructs to build any English and French sentence dealing automatically with most agreements between parts of a sentence (i.e. agreement between the subject and the verb or between different constituents with a noun phrase), a feature that is especially important in French. Most often, number and sometimes gender need to be specified for nouns and tense for verbs only, because other dependents agree automatically. One important feature of JSREALB is its ability to create different forms (e.g negative, passive, modal or interrogative) from a single affirmative structure. It is also possible to generate the pronominal form corresponding to a given noun phrase.

9 Evaluation

As described in Section 7, JSREALB has been used quite extensively in many contexts: in NLG competitions, as the last step for an AMR realizer and in data-to-text applications. The system distribution comes with many unit tests: more than 1000 for English verb conjugation and noun declension and 3000 for French; 100 for English pronouns, dates and numbers and 150 for French. There are also about 100 French and 50 English sentences that test specific *difficult* features in sentences, such as agreement between multiple subjects, pronominalization and sentence transformations.

Generating a sentence is instantaneous needing only a few milliseconds on a commodity laptop; execution time has never been an issue with this realizer. For example in the unit tests, 21 English sentences are processed in 9 milliseconds and 13 milliseconds are enough to realize 38 French sentences.

We never encountered any serious limitation in generating text even when reproducing existing texts (e.g. [Little Riding Hood](#)), but we now describe a more formal evaluation by

sampling sentences found in public corpora and reproducing them with JSREALB, similarly to the methodology used by Braun *et al.* [2] for evaluating SIMPLNLG-DE.

In order to find representative English and French sentences, we randomly sampled ten sentences of more than 5 tokens¹⁵ from the the test corpora of each of the 6 English and 6 French Universal Dependencies 2.7 treebanks [19] for which the morphological features are specified. This gave us 60 English sentences and 60 French sentences. We managed to create JSREALB expressions to reproduce verbatim almost all sentences of our sample. This shows that the coverage of the French and English linguistic phenomena is almost complete. We only encountered a few limitations: JSREALB cannot reproduce contracted forms such as *I'll* for *I will* or *aint* for *is not*, although there are provisions for contractions such as *don't* or *can't*; some word orders cannot be easily obtained, e.g. the insertion of an adverb within a negation as in *you can also remove* that will instead be realized as *you also can remove*; possessives using *'s* cannot be specified although it can be *tricked* by adding the option *.a("'s")* to the possessor's word.

To *regenerate* these 120 sentences, we could have written the JSREALB expressions *from scratch* using a program editor. It takes about two minutes to create an expression for a sentence of about 20 tokens using an [interactive editor embedded in a web page](#) which allows for testing the edited expression.

But, for sentences already annotated with Universal Dependencies, we developed a tool for creating a JSREALB expression using lemma, part of speech and features information [11] (see Figure 8). As the page highlights differences between the original UD text and the sentence realized by JSREALB, it is a simple matter to edit the generated expression to reproduce the original text. This tool is also very useful for checking UD annotations. To our dismay, we found many cases in which the original annotations were incorrect or incomplete: 25 sentences out of our sample of 60 English sentences (42%) and 19 of 60 French sentences (31%) had at least one discrepancy. A detailed analysis of this experiment is presented in [11]. This revealed to be an unexpected use-case: a *naïve* text generator using only the given features can be exploited for debugging annotations. One case is shown in the figure, in which the features (6th field) for token 2 should be `Number=Plur` and not `Sing`, and for token 3 the whole feature field should be `VerbForm=Part|Tense=Pres`.

¹⁵We felt that there is no real NLG interest in generating very short sentences.

Universal Dependencies graph/tree display with regeneration using jsRealB

[Show instructions](#)
[\[Version française\]](#)

```
# comment = with orphaned following dependent due to verb ellipsis in conjunction
# previous = What did the dealers like doing with Alex's cars?
# sent_id = 208
# text = Dealers like seeing cars, theirs especially.
1   Dealers   dealer   NOUN   -   Number=Plur   2   nsubj   -   -   0   root   -
2   like      like     VERB   V   Mood=Ind|Number=Sing|Person=3|Tense=Pres|VerbForm=Fin   0   xcomp   -
3   seeing    see      VERB   V   Mood=Ind|Number=Sing|Person=3|Tense=Pres|VerbForm=Inf   2   -   -
4   cars      car      NOUN   -   Number=Plur   3   obj     -   SpaceAfter=No
5   ,         ,        PUNCT   -   2   punct   -   -
6   theirs    they     PRON   SG-GEN-INDEP   Case=Gen|Gender=Neut|Number=Sing|Person=3|Poss=Yes|PronType=Prs   2
7   especially especially ADV     RB     -   6   orphan  -   SpaceAfter=No
8   .         .        PUNCT   Period   -   2   punct   -   -

# file = en/en_partut-ud-test.conllu
# sent_id = en_partut-ud-997
# text = The term of office of the other Judge shall expire on 31 August 2010.
1   The       the      DET     RD     Definite=Def|PronType=Art   2   det     -   -
2   term      term     NOUN    S      Number=Sing   10  nsubj   -   -
3   of        of       ADP     E      -   4   case    -   -
4   office    office   NOUN    S      Number=Sing   2   nmod    -   -
5   of        of       ADP     E      -   8   case    -   -
6   the       the      DET     RD     Definite=Def|PronType=Art   8   det     -   -
```

Parse 60 sentences ☒ Only differences

Display as ☒ Spacing in pixels: Word 5 Letter 0

UD text	Dealers like seeing cars, theirs especially.
jsRealB	dealers likes see cars, theirs especially.
	2 differences

```
1 CP(VP(N("dealer").n("p"),
2     V("like").t("p").pe("3").n("s"),
3     VP(V("see").t("b").pe("3").n("s"),
4       N("car").n("p"))),
5     NP(Pro("them").c("gen").pe("3").g("n").n("s"),
6     Adv("especially"))).a("."))
```

Figure 8: Web page for creating JSREALB expressions from Universal Dependencies in CONLL-U format that are entered 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. Differences between the expected text and realized text 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.

10 Conclusion

We have described the text realization process of JSREALB an interesting middle ground between a very abstract input specification and a detailed formatting language. It allows automating the *finishing touches* for well-formed language strings displayed to a user. We have shown that the seemingly simple task of producing well-formed natural language text from a relatively abstract formalism involves a lot of intricate language dependent details that must be dealt with, for realizing fluent and syntactically correct sentences.

JSREALB covers the most important basic features of both French and English and comes with comprehensive lexica covering most current uses for which it is straightforward to add new words. The implementation was validated by testing for grammatical functionality, e.g. verb conjugation, and language coverage on sentences taken from UD dependencies.

Acknowledgments

JSREALB was originally developed for French by Nicolas Daoust during his master’s thesis. The bilingual version was then created by Paul Molins and improved by Francis Gauthier during their internship at RALI. Over the years, it has benefited from suggestions by members of the RALI, especially Fabrizio Gotti. We also thank François Lareau from OLST in the Linguistics Department of Université de Montréal for many fruitful discussions and suggestions. Anonymous reviewers on a previous *rejected* journal submission were also very helpful and suggested many improvements and extensions to the original text.

References

- [1] John Bateman. KPML access page. <http://purl.org/net/kpml>, 2016.
- [2] Daniel Braun, Kira Klimt, Daniela Schneider, and Florian Matthes. SimpleNLG-DE: Adapting SimpleNLG 4 to German. In *Proceedings of the 12th International Conference on Natural Language Generation*, pages 415–420, Tokyo, Japan, October–November 2019. Association for Computational Linguistics.
- [3] Thiago Castro-Ferreira, Claire Gardent, Nikolai Ilinykh, Chris van der Lee, Simon Mille, Diego Moussalem, and Anastasia Shimorina. The 2020 Bilingual, Bi-Directional WebNLG+ Shared Task: Overview and Evaluation Results (WebNLG+ 2020). In *Proceedings of the 3rd WebNLG Workshop on Natural Language Generation from the Semantic Web (WebNLG+ 2020)*, Dublin, Ireland (Virtual), 2020. Association for Computational Linguistics.
- [4] Thiago Castro Ferreira, Chris van der Lee, Emiel van Miltenburg, and Emiel Krahmer. Neural data-to-text generation: A comparison between pipeline and end-to-end architectures. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*

- (*EMNLP-IJCNLP*), pages 552–562, Hong Kong, China, November 2019. Association for Computational Linguistics.
- [5] Noam Chomsky. *Syntactic structures*. Mouton de Gruyter, 2nd edition, 2002.
 - [6] Nicolas Daoust and Guy Lapalme. JSreal: A Text Realizer for Web Programming. In N ria Gala, Reinhard Rapp, and Gemma Bel-Enguix, editors, *Language Production, Cognition, and the Lexicon*, number 6636 in Text, Speech and Language Technology, chapter 21, pages 363–378. Springer, jul 2014.
 - [7] Michael Elhadad and Jacques Robin. An overview of SURGE: a reusable comprehensive syntactic realization component. In *Eighth International Natural Language Generation Workshop (Posters and Demonstrations)*, 1996.
 - [8] 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.
 - [9] Guy Lapalme. Realizing Universal Dependencies structures using a symbolic approach. In Simon Mille, Anja Belz, Bernd Bohnet, Yvette Graham, and Leo Wanner, editors, *The Second Multilingual Surface Realisation Shared Task (SR’19): Overview and Evaluation Results. In Proceedings of the 2nd Workshop on Multilingual Surface Realisation (MSR), (EMNLP-2019)*, page 8 pages, Hong-Kong, nov 2019. ACL.
 - [10] Guy Lapalme. Verbalizing AMR structures. <http://rali.iro.umontreal.ca/rali/sites/default/files/publis/GoPhi.pdf>, Aug 2019.
 - [11] Guy Lapalme. Regenerating sentences from Universal Dependencies structures. <http://rali.iro.umontreal.ca/JSrealB/current/demos/UDregenerator/UDregenerator.pdf>, Dec 2020.
 - [12] Fran ois Lareau, Florie Lambrey, Ieva Dubinskaite, Daniel Galarreta-Piquette, and Maryam Nejat. GenDR: A generic deep realizer with complex lexicalization. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan, May 2018. European Language Resources Association (ELRA).
 - [13] Benoit Lavoie and Owen Rainbow. A fast and portable realizer for text generation systems. In *Fifth Conference on Applied Natural Language Processing*, pages 265–268, Washington, DC, USA, March 1997. Association for Computational Linguistics.
 - [14] Susan W. Mcroy, Songsak Channarukul, and Syed S. Ali. An augmented template-based approach to text realization. *Nat. Lang. Eng.*, 9(4):381–420, December 2003.

- [15] Simon Mille, Roberto Carlini, Alicia Burga, and Leo Wanner. FORGe at SemEval-2017 task 9: Deep sentence generation based on a sequence of graph transducers. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 920–923, Vancouver, Canada, August 2017. Association for Computational Linguistics.
- [16] 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.
- [17] Kees van Deemter, Emiel Krahmer, and Mariët Theune. Squibs and discussions: Real versus template-based natural language generation: A false opposition? *Computational Linguistics*, 31(1):15–24, 2005.
- [18] Pierre-Luc Vaudry and Guy Lapalme. Adapting SimpleNLG for bilingual English-French realisation. In *14th European Conference on Natural Language Generation*, pages 183–187, Sofia, Bulgaria, Aug 2013.
- [19] Daniel Zeman, Joakim Nivre, Michell Abrams, and *et al.* Universal Dependencies 2.7. <http://hdl.handle.net/11234/1-3424>, nov 2020.