RDFjsRealB: a Symbolic Approach for Generating Text from RDF Triples

Guy Lapalme

November 17, 2020

Abstract

This paper describes the Resource Description Framework (RDF) triples verbalizer developed for the WebNLG Challenge 2020 shared task. After reviewing representative works in Natural Language Generation in the context of the Semantic Web, the task is then described. We then sketch the symbolic approach we used for verbalizing RDF triples: once the triples are grouped by subject, each group is realized as one or more sentences using templates written in Python whose output is feed to an English realizer written in Javascript. The system was developed using the test data of the previous edition of the task and the train and development data of this year's task. The automatic scores for this year's test data are quite competitive. We conclude with a critical review of the data and discuss the suitability of this competition results in a wider Natural Language Generation setting.

1 Context

This paper describes our system developed for participating to the *RDF-to-text generation for English* subtask of the WebNLG Challenge 2020 Castro-Ferreira et al. (2020) as a follow-up to 2017 edition (Gardent et al., 2017b). The WebNLG Challenge 2020 data was developed for pushing the development of Resource Description Framework (RDF) verbalizers for realizing short texts while dealing with some micro-planning problems such as sentence segmentation and ordering, referring expression generation and aggregation. The first edition of the data featuring 15 DBpedia categories was created in 2017 (Gardent et al., 2017a). The 2020 challenge[1] covers more categories and an additional language, Russian and a new task: Text-to-RDF semantic parsing for converting a text into the corresponding set of RDF triples.

Our English RDF verbalizer is based on a symbolic approach using Python: each RDF triple corresponds to a sentence in which the subject and the object of a triple are mapped almost verbatim as subject and object of the sentence. The predicate of the triple corresponds

[1] `https://webnlg-challenge.loria.fr/challenge_2020/`

to a verb phrase which determines the structure of the sentence. The predicates are ordered to create a meaningful story and parts of sentences are merged when they share subjects or predicates. The final realization is performed using jsRealB[2], a French-English realizer that we upgraded in recent years.

2 Related work

Generating text from ontologies and RDF has a long history, because it is a typical case of computer-oriented data about entities that need to be explained to a human in order to be understood or modified. Bouayad-Agha et al. (2014) present a comprehensive discussion of Semantic Web concepts and its relation with Natural Language Generation. One challenge of RDF verbalization is the fact that the information is spread over a graph linking many entities that must be linearized as a coherent text. RDF triples cannot be *lexicalized* directly as they need links with other information in order to be conveyed correctly. Fortunately, most well-organized RDF triples depend on an ontology, a hierarchy of concepts, that allows inferencing to help drive the generation process.

Some systems target the generation of texts for explaining ontologies while other use ontologies for explaining facts expressed in RDF which reflects two trends in the Semantic Web area: ontology construction using the Web Ontology Language (OWL) with an emphasis on the logical consistency, or the publication of a large number of linked data without necessarily ensuring consistency. WebNLG Challenge 2020 targets the latter because it is limited to the verbalization of a few RDF triples, but we think it is interesting to briefly look at the former, because it gives a broader view of potential applications for future developments.

2.1 Explaining or Using Ontologies

Aguado et al. (1998) motivate the use of text generation for explaining ontologies to help their reuse. They illustrate their approach by generating Spanish explanations in the domain of chemical substances. They combine the *General Upper Model* (GUM) (Bateman et al., 1995) approach with the KPML (Bateman, 1997) text realizer. Wilcock and Jokinen (2003) use the information in the ontology as background information for a dialogue system that provides information about a public transportation system. The ontology serves both as a source of information and for identifying misconceptions and suggesting alternative reasonable questions. Bontcheva and Wilks (2004) show how to generate reports from domain ontologies; they present a use case in the area of breast cancer in which the concepts of the ontology were manually mapped to words of a specialized lexicon.

Galanis et al. (2009) describe NaturalOWL[3], a plug-in for the Protégé[4] ontology editor that produces template-based descriptions of entities and classes from OWL ontologies

[2]See `https://github.com/rali-udem/jsRealB` for documentation, a tutorial and examples of use.

[3]`https://protegewiki.stanford.edu/wiki/NaturalOWL`

[4]`https://protege.stanford.edu`

that have been annotated with linguistic and user modeling information expressed in RDF. Given that it is open-source and embedded in a widely used ontology editor, it has been used as a baseline in many subsequent works.

2.2 Verbalizing RDF statements

The first step in generating texts from RDF is finding an appropriate subset of RDF statements. Duboue and McKeown (2003) were pioneers in determining relevant content for NLG using statistical methods for the extraction of facts from texts and coupling them with the associated data. This approach inspired most recent learning methods given the availability of a large number of texts and associated data in DBPedia and Wikipedia. Duboue and McKeown had to parse HTML pages and databases to find a large set of biographies. For WebNLG Challenge 2020, this essential but difficult step has already been done by the organizers of the competition (Gardent et al., 2017a).

Sun and Mellish (2006) take advantage of the fact that RDF representations are not only logical representations, but that they also contain rich linguistic information useful for generating text. After studying many published ontologies, they found systematic patterns in class and relation names that can be exploited for lexicalization without developing special purpose dictionaries.

Duma and Klein (2013) propose a system that can automatically learn sentence templates and document planning from parallel RDF data and text from the Simple English Wikipedia. They first match named entities in the sentence with a graph related to a specific entity in order to extract a template in which named entities are replaced by a variable. To prune entities and their dependents in the sentence not appearing in the graph, the sentences are first parsed and a few hand-written rules operating on the syntactic tree are applied as suggested by Gagnon and Da Sylva (2006) for summarization purposes. The content selection uses the method originally suggested by Duboue and McKeown. To determine relevant predicates, they determine a *prototypical* class by looking at the most frequent *subwords* in the class names which often use *camelCase*. Given the URI of an entity to be described, they determine the relevant class and its associated templates that are used for creating many sentences from which the best ones are chosen.

Ell and Harth (2014) show how to extract verbalization templates for RDF graphs from DBPedia and the corresponding Wikipedia documents about specific types of entities. Sentences that mention the entities are aligned with a data graph using language-independent transformations. The connected entities in the graph are then iteratively explored to find commonalities that allow some abstraction of the entities using variables. They thus obtain a set of abstracted sentences from which templates are created. They applied their technique on English and German with *promising* results.

Dong and Holder (2014) present *Natural Language Generation from Graphs* (NLGG) with three processing stages: model preparation and content determination, document structuring, and lexicalization, aggregation and realization to create English text. It uses templates that code linguistic information about each class such as its English label both singular and plural; the relations are also coded with templates that indicate the type of its subject and

object and priority to drive the text organization. Model preparation uses an RDF reasoner to infer new triples and remove redundant ones. Document structuring consists in deciding the order of output: first classes, then attributes and finally relationship information. SimpleNLG (Gatt and Reiter, 2009) is used for creating the English text. Our system follows a similar approach.

Vougiouklis et al. (2018) describe a statistical model for NLG by adapting the encoder–decoder framework to generate textual summaries for triples related to biographies. They use sequence to sequence methods to jointly perform content selection and surface realization without any rules or templates. Since triples are not sequentially correlated, they develop a feed-forward neural network that encodes each triple into a vector of fixed dimensionality in a continuous semantic space in which triples having similar semantic meaning have similar positions. This encoder is coupled with an RNN-based decoder that generates the textual summary one token at a time. They conducted many evaluations and discuss openly the limitations of their original work: results are best with fewer than 10 triples, some repetitions often occur and sentences can also state frequent facts in the database but that do not occur in the input triples.

Gardent et al. (2017b) present the results of the WebNLG Challenge 2017 that compared the output produced by 8 submissions by 6 teams: three submissions used a template or grammar-based pipeline framework combined with a symbolic realizer; this is the approach that we present in this paper. One system used a statistical machine translation framework and four submissions used an attention-based encoder-decoder architecture built using existing neural machine translation frameworks. Comparisons with our system will be discussed in Section 5.

Zhu et al. (2019) propose a way of improving the quality of the text at the expense of diversity by optimizing the inverse KL divergence for conditional language generation. Their paper presents a detailed discussion on the fundamental problems of minimizing KL divergence in training for this problem and justify the inverse KL divergence as their optimization objective. They experimented on three datasets, one of which was WebNLG Challenge 2017. They managed to get somewhat better automatic scores over alternative techniques. They also showed the best results in human evaluation for grammar and consistency on 20 triples.

We now give some details on the input for the competition and then we will now present the approach that we use for WebNLG Challenge 2020.

3 Data for WebNLG Challenge 2020

Table 1 states some facts about the astronaut Alan Shepard; this example is from the *dev set* of the competition data. The top part shows 12 RDF triples which could be the *raw input* for an RDF realizer. They were extracted out of the billions in DBPedia by first selecting N-Triples downloaded from `http://dbpedia.org/page/Alan_Shepard` which resulted in 235 triples having *Alan Shepard* as subject. Other triples about *Apollo 14*, *New Hampshire*, *California* and *NASA* were then similarly added.

```
<http://dbpedia.org/resource/Alan_Shepard> <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://dbpedia.org/ontology/Astronaut> .

<http://dbpedia.org/resource/Alan_Shepard> <http://www.w3.org/2000/01/rdf-schema#label>
    "Alan␣Shepard@en" .
<http://dbpedia.org/resource/Alan_Shepard> <http://dbpedia.org/ontology/mission>
    <http://dbpedia.org/resource/Apollo_14> .
<http://dbpedia.org/resource/Apollo_14> <http://www.w3.org/2000/01/rdf-schema#label>
   "Apollo␣14"@en .

<http://dbpedia.org/resource/Alan_Shepard> <http://dbpedia.org/ontology/deathPlace>
    <http://dbpedia.org/resource/California> .
<http://dbpedia.org/resource/California> <http://www.w3.org/2000/01/rdf-schema#label>
    "California"@en .

<http://dbpedia.org/resource/Alan_Shepard> <http://dbpedia.org/ontology/birthPlace>
    <http://dbpedia.org/resource/New_Hampshire> .
<http://dbpedia.org/resource/New_Hampshire> <http://www.w3.org/2000/01/rdf-schema#label>
   "New␣Hampshire"@en .

<http://dbpedia.org/resource/Alan_Shepard> <http://dbpedia.org/property/dateOfRet>
    "1974-08-01"^^<http://www.w3.org/2001/XMLSchema#date> .

<http://dbpedia.org/resource/Apollo_14> <http://dbpedia.org/property/operator>
    <http://dbpedia.org/resource/NASA> .
<http://dbpedia.org/resource/NASA> <http://www.w3.org/2000/01/rdf-schema#label> "NASA"@en .

<http://dbpedia.org/resource/Alan_Shepard> <http://dbpedia.org/ontology/birthDate>
    "1923-11-18"^^<http://www.w3.org/2001/XMLSchema#date> .
```

```
<entry category="Astronaut" eid="Id4" ... size="6">
  <originaltripleset>
    <otriple>Alan_Shepard | mission | Apollo_14</otriple>
    <otriple>Alan_Shepard | deathPlace | California</otriple>
    <otriple>Alan_Shepard | birthPlace | New_Hampshire</otriple>
    <otriple>Alan_Shepard | dateOfRet | "1974-08-01"^^xsd:date</otriple>
    <otriple>Apollo_14 | operator | NASA</otriple>
    <otriple>Alan_Shepard | birthDate | "1923-11-18"^^xsd:date</otriple>
  </originaltripleset>
  <modifiedtripleset>
    <mtriple>Alan_Shepard | mission | Apollo_14</mtriple>
    <mtriple>Alan_Shepard | deathPlace | California</mtriple>
    <mtriple>Alan_Shepard | birthPlace | New_Hampshire</mtriple>
    <mtriple>Alan_Shepard | dateOfRetirement | "1974-08-01"</mtriple>
    <mtriple>Apollo_14 | operator | NASA</mtriple>
    <mtriple>Alan_Shepard | birthDate | "1923-11-18"</mtriple>
  </modifiedtripleset>
  <lex comment="good" lid="Id1">Alan Shepard was born on November 18, 1923, in New
   Hampshire. He served as a crew member on NASA's Apollo 14 mission before retiring
   on August 1, 1974. Shepard passed away in California.</lex>
  <lex comment="good" lid="Id2">Alan Shepard was part of NASA's Apollo 14
  crew. He was born in New Hampshire on November 18th, 1923, he retired
  August 1st 1974 and died in California.</lex>
  <lex comment="good" lid="Id3">Alan Shepard was born in 1923 in New Hampshire. He worked
  for NASA and became a crew member for Apollo 14 before he retired in August 1974.
  He later died in California.</lex>
</entry>
```

Alan Shepard was born on November 18, 1923 in New Hampshire and he was a crew member of Apollo 14 that is operated by NASA. He went into retirement on August 1, 1974 and passed away in California.

Table 1: The top part shows 12 triples extracted from DBPedia used to create the XML input from `/dev/en/6triples/Astronaut.xml` shown on the middle part. The bottom part shows the realized sentence produced by RDFJSREALB from this input.

For WebNLG Challenge 2017, these triples were selected using an elaborate content selection method described by Perez-Beltrachini et al. (2016) which, given some DBPedia entity, retrieves DBPedia subgraphs that encode relevant and coherent knowledge about that entity.

We recall that an RDF triple is composed of three URIs corresponding to the subject, the predicate and the object which can also be a constant string, a date or a number. The predicate of a triple declares a relation between the subject and the object, such as `Alan Shepard - birthPlace - New Hampshire`, in which `Alan Shepard` is the subject, `birthPlace` the predicate indicating that the subject was born at the place given by the object and `New Hampshire` is the object. This could be verbalized as Alan Shepard is born in New Hampshire. A set of triples corresponds to a graph that can be encoded using many formalisms. The appendix 8 shows the rendition of these triples in RDF-XML and Turtle (Beckett et al., 2014) with the corresponding graph.

The triples at the top part of Table 1 are separated by blank lines into 7 groups. The first triple indicates the category associated with the selected topic, `Astronaut` in this case. RDF triples are computer-friendly unique URIs, but are not very convenient for NLP, so the corpus creators used instead their associated English labels. For example, the second group (separated by an empty line) of three triples were combined to create the three components (separated by a vertical bar) of the first `otriple` element in the middle part of the table. The next five groups of triples were used to create the other `otriple` elements of the `originaltripleset`.

Gardent et al. (2017a) explain how these triples were then *normalized*, using a combination of automatic and manual processes, to create the `modifiedtripleset` that is used as input for the competitions. The `lex` elements that give human renditions of the triples were obtained through a crowdsourcing process. The content of these elements is used for training systems and evaluation. We used them as inspiration for manually writing the templates used by RDFjsRealB, but after writing almost 200 of them, we are contemplating using machine learning techniques to do this.

The English WebNLG Challenge 2020 dataset for English has an impressive number of data-text pairs: 13 211 set of triples coupled with 35 426 texts for training, and 1 667 sets of triples linked with 4 464 texts for development. They come from 16 categories (e.g. *Airport, Artist, Astronaut...*) and are split according to the number of triples they contain (between 1 and 7). Some triples are repeated, in fact, all triples that occur at least once in the corpus appear as one of the 3 600 *1_triples*. The Corpus documentation[5] states: *Theoretically, that hierarchical corpus construction enables to produce texts expressing 7 triples by using only 1_triple entries.*

Of course, combination with other triples will change the realization of a given triple when used alone. In the training corpus, there are 372 different predicates and 2926 different subjects or objects (ignoring numbers).

[5] `https://webnlg-challenge.loria.fr/docs/`

3.1 External data

We developed a first version of the system using only triples as input to the system, but after a preliminary examination of the results, we felt that we needed some more context. As the WebNLG Challenge 2020 competition allows external resources, we developed two functions to query the DBPedia SPARQL endpoint[6] for :

- checking if a *subject* corresponds to a given *category* using :

```
ASK WHERE { <http://dbpedia.org/resource/subject>
            rdf:type
            <http://dbpedia.org/ontology/category> }
```

- getting the gender of a subject:

```
SELECT ?gender
   WHERE { <http://dbpedia.org/resource/subject>
           <http://xmlns.com/foaf/0.1/gender>
           ?gender }
```

The above SPARQL queries are sent using the Python interface SPARQLWrapper.

4 Text Generation

The first step in text generation is determining the information to be conveyed in the text. In the context of WebNLG Challenge 2020, this is given: it is a set of at most 7 triples. As the predicate of a triple indicates a relation between its subject and object, in our case, it is mapped to a verb linking the subject and the object of the sentence realizing to this triple.

4.1 Microplanning

Triples being unordered, the first critical step is organizing them to build an *interesting story*. The triples are first grouped by their subject and the triples are sorted within their group. For example, to describe a person, the birth date and place could first be given, then some activities, finally the retirement and death; for a University or a company, first its creation date, then its activity. To achieve this ordering, we associate with each predicate a *priority* used for sorting the input triples. We also submitted for automatic scoring a version of the system that skipped this sorting process. Although the automatic scores (see Section 5) were highly similar for both versions, we conjecture the sorting process is still useful to make the texts easier to follow.

The sorted groups are then processed in decreasing number of triples. If the category of a group subject is the current one (using the function described in Section 3.1), then its score is increased so that the text starts with this subject similarly to what we saw in

[6] http://dbpedia.org/sparql

```
Alan_Shepard
   birthDate "1923-11-18";
   birthPlace New_Hampshire;
   mission Apollo_14;
   dateOfRetirement "1974-08-01";
   deathPlace California.
Apollo_14
   operator NASA.
```

Table 2: `mtriple`s from Table 1 sorted and grouped, shown as a Turtle-like formalism, used as input for RDFjsRealB. Predicates and objects sharing the same subject are shown indented and separated by semicolons.

the lexicalizations in the training corpus. Each group forms a sentence as a coordination of *subsentences*. As a long coordinated sentence is often difficult to follow, groups of more than 3 triples are split into two sentences. In order to avoid very short sentences, groups with only one triple are combined using a subordinate when its subject is the object of another triple in a bigger group.

Table 2 shows the result of the sorting and grouping process on the example of Table 1. The five triples having `Alan_Shepard` as subject are grouped and sorted to form a coherent biography. This input will be used for realizing the two sentences shown in the bottom part of Table 1 using jsRealB.

4.2 Surface realization

For the final realization step, we use jsRealB (Molins and Lapalme, 2015), a surface realizer written in Javascript[7] similar in principle to SimpleNLG (Gatt and Reiter, 2009) in which programming language instructions create data structures corresponding to the constituents of the sentence to be produced. Once the data structure is built, it is traversed to produce the list of words in the sentence, dealing with conjugation, agreement, capitalization, all the *small* details that are important for easing the reading by the users and evaluators. Unfortunately, a large part of this hard work is not taken into account by the automatic evaluation process which often works with lowercased tokens.

For RDFjsRealB, we use Python to create the structure sent to a local jsRealB web server that returns the realized sentence. The data structure is built by calls to constructors whose names were chosen to be similar to the symbols typically used for constituent syntax trees for the complete list of functions and parameter types:

- **Terminal**: `N` (Noun), `V` (Verb), `A` (adjective), `D` (determiner) ... `Q` quotes its parameter thus allowing *canned text*.
- **Phrase**: `S` (Sentence), `NP` (Noun Phrase), `VP` (Verb Phrase) ...

[7]See the documentation `http://rali.iro.umontreal.ca/JSrealB/current/documentation/user.html?lang=en`

```
S(Q("Alan␣Shepard"),                  # quoted string
  VP(V("be").t("ps"),                 # auxiliary to the simple past
     V("born").t("pp"),               # verb to past participle
     PP(P("on"),                      # prepositional phrase
        DT("1923-11-18").dOpt(...)), # date (options are not shown here)
     PP(P("in"),                      # prepositional phrase
        Q("New␣Hampshire"))))         # another quoted string
```

```
{"phrase":"S",
 "elements":[{"terminal":"Q","lemma":"Alan␣Shepard"},
             {"phrase":"VP",
              "elements":[{"terminal":"V","lemma":"be","props":{"t":"ps"}},
                          {"terminal":"V","lemma":"born","props":{"t":"pp"}},
                          {"phrase":"PP",
                           "elements":[{"terminal":"P","lemma":"on"},
                                       {"terminal":"DT","lemma":"1923-11-18"}]},
                          {"phrase":"PP",
                           "elements":[{"terminal":"P","lemma":"in"},
                                       {"terminal":"Q","lemma":"New␣Hampshire"}]}]}]}
```

Table 3: Top: Python/JavaScript functional notation for a jsRealB expression with comments at the right. Bottom: JSON notation equivalent to the top. Both are realized by jsRealB as: Alan Shepard was born on November 18, 1923 in New Hampshire.

Features added to the structures using the dot notation 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, these features were not used here, but we use the automatic processing of coordinated phrases that insert appropriate commas and conjunction between coordinated elements.

Table 3, shows the Python calls to create an internal structure that is serialized and sent to the jsRealB server to get an English sentence[8] .

4.3 Sentence Templates

The challenge is thus to transform the structure described in Table 2 to the one in Table 3. We manually defined 200 templates associated with the most frequent predicates in the training set (i.e., those with 20 or more occurrences). A default template is used when defined template is found in less than 10% of cases (see Table 6), but this happens in more than 20% of cases in the test set.

A predicate `p` corresponds to a Python lambda expression whose parameter is the object `o`. The predicate is called to create a sentence with the subject `s`. The actual parameters are quoted strings of the subject or object of the triple, but replacing underscores by spaces with special cases for numbers and dates. For example, given the two following Python definitions:

```
birthPlace = lambda o:VP(V("be").t("ps"),V("born").t("pp"),PP(P("in"),o))
sentence   = lambda s,p,o: S(Q(s),
                             p(Q(o)))
```

[8]With this notation in Python, it would be interesting to develop pyRealB that would realize the sentence in Python, but we leave this development as an exercise for the reader...

calling `sentence("Alan␣Shepard",birthPlace,"New␣Hampshire")` creates the following structure:

```
S(Q("Alan␣Shepard"),
  VP(V("be").t("ps"),V("born").t("pp"),PP(P("in"),Q("New␣Hampshire"))))
```

which is verbalized as Alan Shepard was born in New Hampshire. by jsRealB. This is the basic mechanism for creating sentence structures that can be combined in various ways.

Templates are organized in a dictionary having the name of the predicate as a key associated with a value which is a 3-tuple with the following elements:

- its priority (a number between 0 and 100) that is used for sorting the predicates; currently, priorities are given using *rules of thumb*, things that we felt should be said at the beginning of a text (e.g. birth date) are given small numbers and bigger numbers for informations to be given at the end (e.g. death date) otherwise 50 is assigned.
- a boolean indicating if its subject can be a human;
- a list of lambda expressions that can verbalize this predicate, one of which is randomly chosen at the realization time.

Here are a few predicates out of the nearly 200 that were developed by looking at `lex` elements in the training corpus. When two templates have the same realizations, the third element of the pair is the name of the original predicate (see `birthYear` below).

```
sentencePatterns = {
  ...
  "birthDate":(2,True,[
    lambda o:VP(V("be").t("ps"),V("born").t("pp"),PP(P("on"),o)),
  ]),
  "birthPlace":(3,True,[
    lambda o:VP(V("be").t("ps"),V("born").t("pp"),PP(P("in"),o)),
  ]),
  "birthYear":(2,True,"birthPlace"),
  "dateOfRetirement":(90,True,[
    lambda o:VP(V("retire").t("ps"),PP(P("on"),o)),
    lambda o:VP(V("go").t("ps"),P("into"),N("retirement"),PP(P("on"),o)),
  ]),
  "deathPlace":(100,True,[
    lambda o:VP(V("die").t("ps"),PP(P("in"),o)),
    lambda o:VP(V("pass").t("ps"),Adv("away"),PP(P("in"),o)),
  ]),
  "mission":(22,True,[
    lambda o:VP(V("be").t("ps"),D("a"),N("crew"),N("member"),PP(P("of"),o)
       ),
    lambda o:VP(V("become").t("ps"),N("member"),PP(P("of"),o)),
  ]),
  "operator":(51,False,[
    lambda o:VP(V("be"),V("operate").t("pp"),PP(P("by"),o)),
  ]),
  ...
}
```

Currently a template only depends on the name of the predicate, but it would be interesting to develop specialized templates depending on the category of the set of triples; for example, `language` used for a `WrittenWork` should be verbalized as written in, but for an `Artist` or a `Politician` it could be speaks or sings in.

Once the above structure for the templates was settled after a few false starts, it became relatively easy to write them. Reading lexicalizations associated with a predicate, it takes less than minute to write a lambda with a jsRealB constituent expression to reproduce of them. This is possible because we noticed that many lexicalizations are often very similar, having been created by crowdworkers who seemed to often rely on copy-pasting the subject and the object. We conjecture that it should be feasible to develop a learning algorithm to go from the lexicalizations to the lambdas, the final realization being left to jsRealB.

4.4 Default template

When an unknown predicate is encountered then a default template is created. By detecting case changes, the name of the predicate is split into *words*. and taken as subject of the be auxiliary, the object is used as attribute. For example,

```
elevationAboveTheSeaLevel => Q("elevation␣above␣the␣sea␣level")
```

In the final sentence, the subject of the triple is taken as subject of the be auxiliary, the object of the triple is used as attribute. For example, the triple

```
Aarhus_Airport | elevationAboveTheSeaLevel | 25.0
```

is realized as Aarhus Airport elevation above the sea level is 25.0.

4.5 Text aggregation

In some cases, dealing with related information (e.g., birth date and place), combining templates using only their complements (i.e., their last element) will simplify the text. For this we define groups of predicates that can be combined at realization time. For example, `birthDate` and `birthPlace` shown in the previous listing or
`numberOfStudents`, `academicStaffSize` and `numberOfPostgraduateStudents`. To this list are added, the equivalent templates in the `sentencePatterns` (e.g. `birthYear` will be combined with `birthDate`). A similar process is used for combining objects sharing their subject and predicate. The proper generation of a coordinated is handled by jsRealB : in a coordinated phrase `CP` (see Table 4), it searches for the conjunction `C` and adds it between the last two coordinated constituents, the previous constituents being separated by commas.

When two or three triples are merged into a single sentence, the subject is used at the start but a pronoun is used for the following references. Currently, a very simple system is used for choosing the pronoun: if the predicate is coded as being applicable to a human and the gender of the subject[9] is `male`, he is used, if it is `female` then she [10] is chosen, otherwise

[9] given by the function described in Section 3.1

[10] There are very few references to a woman in the training set. A rough estimate: she occurs 119 times in the `lex` elements, one of these being a ship, while he has 2,808 occurrences (24 times more). A classical

it is used. When a single triple whose subject is used as object of another, it is combined with the subordinate using a pronoun, who if the predicate applies to a human, otherwise that is used.

Table 4 shows the result of combining all these processes on the input of Table 2.

case of gender-bias inferred from the data. This ratio is quite different in the test set in which there are 669 references to a female *against* 768 to a male. Counting occurrences in the same way in the training set, there are 8,855 males and 900 females, while in the development set there are 1,086 males and 130 females (a ratio of about 10).

```
S(CP(C("and"),                # coordination
     S(Q("Alan␣Shepard"),     # first part of the coordination
       VP(V("be").t("ps"),
          V("born").t("pp"),
          PP(P("on"),          # combination of objects
             DT("1923-11-18").dOpt(...)),
          PP(P("in"),
             Q("New␣Hampshire")))),
     S(Pro("I").g("m"),       # second part of the coordination
       VP(V("be").t("ps"),
          D("a"),
          N("crew"),
          N("member"),
          PP(P("of"),
             NP(Q("Apollo␣14"),
                SP(Pro("that"), # integration of a single triple
                   VP(V("be"),
                      V("operate").t("pp"),
                      PP(P("by"),
                         Q("NASA")))))))))
# Alan Shepard was born on November 18, 1923 in New Hampshire
# and he was a crew member of Apollo 14 that is operated by NASA.

S(CP(C("and"),                # coordination
     S(Pro("I").g("m"),       # first part
       VP(V("retire").t("ps"),
          PP(P("on"),
             DT("1974-08-01").dOpt(...)))),
     S(Q(""),                 # second part
       VP(V("pass").t("ps"),
          Adv("away"),
          PP(P("in"),
             Q("California"))))))
# He retired on August 1, 1974 and passed away in California.
```

Table 4: Two indented structures produced by aRDFJSREALB Python program for the example of Table 1 annotated here with Python comments. Each sentence structure is followed by a comment showing the structure realization produced by JSREALB.

System	BLEU	METEOR
RDFjsRealB	0.41	0.40
2017-best	0.45	0.39
2017-baseline	0.33	0.23
2017-worst	0.07	0.09

Table 5: Comparison on two automatic score values on the WebNLG Challenge 2017 test set. The top line shows the score of RDFjsRealB and the bottom part shows the range of scores by 8 systems that participated to WebNLG Challenge 2017.

5 Results

RDFjsRealB has been applied to the test set of WebNLG Challenge 2017 and to the dev and train set of WebNLG Challenge 2020. The system is very fast, it processes about 1 000 triple sets per second on Mac laptop; this is much faster than the evaluation which takes many seconds to process a single category with a given number of triples. Table 5 shows the scores on the WebNLG Challenge 2017 test set. Table 6 shows that the scores on the WebNLG Challenge 2020 test set are quite competitive. We only give the values of the BLEU score, the others being strongly correlated with these .

We submitted three versions of our system for the automatic scoring WebNLG Challenge 2020:

1. RDFjsRealB: as described in the preceding sections;

2. RDFjsRealB-unsorted: but skipping sorting the predicates that share a subject. We wanted to see the effect of ignoring the priorities which seemed a bit artificial;

3. RDFjsRealB-baseline: creates a sentence by concatenating the subject, the words contained in the predicate split as we do for the default template (Section 4.4) and the object of a triple. Sentences are then concatenated to create the text.

RDFjsRealB is quite competitive: while being far from the best, it gives very good results being in the first third of 35 participants. It is roughly at par with the competition baseline which is also a symbolic system. We see that the sorting process improves scores by a very small margin for automatic scoring. We are curious to see how it will influence the human scoring. As expected, our baseline (a 10 line Python program shown in Appendix 8) is much worse, but surprisingly it is not the worst of all submissions. We further discuss the interest of this submission in the next section.

For WebNLG Challenge 2020 automatic scoring, we submitted three versions: one implementing all features we described above and another that ignored the sorting according to the priorities. We see that at least for automatic scoring, this sorting process does not have a significant influence on the results. While during development, the default template was used less than 10% of the time, it was used 24% of the time during the test. There were

2017 test: 5,378 triples						
N	#Triples	Def. tmp	BLEU	B.NLTK	METEOR	chrF++
1-5	5378	7 %	0.41	0.39	0.40	0.64
2020 dev: 4,841 triples						
N	#Triples	Def. tmp	BLEU	B.NLTK	METEOR	chrF++
1	403	8 %	0.52	0.51	0.45	0.74
2	626	7 %	0.47	0.45	0.42	0.68
3	1038	6 %	0.40	0.39	0.39	0.64
4	1280	4 %	0.41	0.40	0.40	0.65
5	1190	5 %	0.37	0.36	0.38	0.62
6	150	14 %	0.39	0.38	0.38	0.63
7	154	10 %	0.33	0.33	0.36	0.61
2020 train: 38,399 triples						
N	#Triples	Def. tmp	BLEU	B.NLTK	METEOR	chrF++
1	3195	9 %	0.55	0.54	0.46	0.74
2	4944	5 %	0.46	0.45	0.42	0.68
3	8205	5 %	0.40	0.39	0.40	0.64
4	10196	5 %	0.39	0.37	0.39	0.63
5	9395	5 %	0.36	0.35	0.38	0.61
6	1218	7 %	0.37	0.36	0.38	0.63
7	1246	8 %	0.35	0.34	0.36	0.61

Table 6: Automatic evaluation results: average of scores for all categories for 4 similarity metrics between the output of RDFJSREALB and up to three references. The first column shows the number of triples in each set, the second, the total number of triples. The third column shows the percentage of times the default template was used. The last four columns show the scores: BLEU (divided by 100 to get numbers in the same range as for the other metrics), BLEU NLTK, Meteor and chrF++ (character n-gram F-score) (Popović, 2015) as computed by the evaluation package provided by the organizers of the shared task.

	All		Seen-Cat		Unseen-Cat		Unseen-Ent	
System	MET	rank	MET	rank	MET	rank	MET	rank
2020-best	0.42	1	0.44	1	0.40	1	0.42	1
RDFjsRealB	0.39	11	0.39	23	0.38	12	0.40	12
RDFjsRealB-unsorted	0.38	12	0.39	25	0.37	14	0.40	11
2020-baseline	0.37	16	0.39	28	0.36	15	0.38	15
RDFjsRealB-baseline	0.33	27	0.33	34	0.33	19	0.32	23
2020-worst	0.22	35	0.33	35	0.13	35	0,30	35

Table 7: METEOR scores and rank among the 35 submissions to WebNLG Challenge 2020 for three versions of RDFjsRealB. We only report METEOR scores as other metrics seem quite correlated with them.

78 unseen templates for the tests of which 30 were used more than 10 times. So we estimate that it would have taken about 30 minutes to develop new templates for dealing with these new cases and improve the results.

5.1 Human Evaluation

At the time of writing this paper, human evaluation results for WebNLG Challenge 2020, were not yet published.

During system development, to get a sense of the quality of the output, a colleague and I evaluated manually the output of a random sample of 100 sets of triples from the training set using the crude scale shown in Table 8. As we had noticed that 1-Triples were almost always correct, that they accounted for about 30% of the set, that they did not raise any interesting NLG challenges and that they are useless for all practical purposes, we decided not to take them into account in the sampling process.[11] So we sampled only between the 2-Triples to the 7-Triples. Table 9 shows good results (3.29 on average) across of tuples of all lengths. There are very few 6 or 7-Triples, this reflects the overall distribution of the triple sets for which there are only 4 categories instead of 16 for the others.

4	Perfect translation of all triples and acceptable English formulation
3	Translation correct, but bad English formulation
2	Translation correct, but English barely understandable
1	Gibberish in the English or missing important information from the triples

Table 8: Scale used for the evaluation of the results

[11] Unfortunately, 20% of the triple sets in the test corpus are 1-Triples.

nTriples	Avg	Nb
2	3.53	18
3	3.17	33
4	3.08	24
5	3.39	19
6	4.00	1
7	3.60	5
Avg	3.29	100

Table 9: Results of the human evaluation. The first column shows the number of triples in the set, the second column the average score for the number of sets shown in the third column. Triples were evaluated by two persons, one of which is the author. The score in the table is the average of these two scores.

6 Comments on the task data

We are very thankful of the task organizers who spent an enormous amount of time and energy building this corpus and collecting human lexicalizations. We also recognize the fact that they explicitly say that their goal was to provide enough data so that it becomes feasible for learning algorithms to develop micro-planners.

We now take a step back to reflect on how this task corresponds to the original motivation: providing verbalizers for *real* RDF set of triples, a goal stated in almost all papers cited in Section 2. We think that WebNLG Challenge 2020 has so greatly simplified the task that many of the *interesting* problems have been more or less bypassed.

As the data already identifies the triples to verbalize, the challenging step of searching an RDF triple store to select the appropriate statements is short-circuited. The problem is thus greatly simplified, but there is still plenty of interesting work to do, so this is not a fundamental criticism, as this step could be performed by another system or could be the subject of another shared task.

An important problem in NLG is lexicalization, i.e., finding the appropriate words to use for an utterance. In WebNLG Challenge 2020, this problem is greatly simplified by the fact that the URIs for the subject and object have been replaced by their labels, see Table 1. So, in almost all cases, it is sufficient just to copy the subject and the object in the output like we do in RDFjsRealB. The only lexicalization problem left is thus finding an appropriate wording for the predicate.

To illustrate how the reference sentences are similar to content of the triples, we developed a simplistic baseline which creates a sentence concatenating the subject, the words contained in the predicate (split as we do for the default template Section 4.4) and the object of a triple. Sentences are then concatenated to create the text. Table 10 shows a surprisingly high similarity for 1-Triples (BLEU scores of more than 30). Of course, similarity decreases as the number of triples increases.

A commonly used convention for naming a predicate is to use a conjugated verb (Uschold, 2018, p. 187) (e.g. `works` or `speaks`) or a verbal locution with words joined using *camelCase*

N	dev	train
1	0.31	0.34
2	0.27	0.25
3	0.22	0.23
4	0.25	0.24
5	0.21	0.22
6	0.25	0.21
7	0.18	0.19

Table 10: Average of BLEU scores obtained using a simplistic baseline.

(e.g. `isPartOf` or `hasChild`). Unfortunately, very few of the predicates in WebNLG Challenge 2020 data follow this convention which is widely used in ontology works. For example, here are the most frequent predicates in the train corpus: `country`, `leader`, `location`, `birthPlace`, `isPartOf`, `club`, `ethnicGroup`, `language`, `genre`, `capital`.

There are also many *long-winded* names for predicates such as

```
addedToTheNationalRegisterOfHistoricPlaces
elevationAboveTheSeaLevelInMetres
wasGivenTheTechnicalCampusStatusBy
```

which should have been divided into two or three triples to separate the operation (`added` or `isElevated`) from the destination or the comparison and the units. A similar remark can be done for `hasToItsNorth`, `hasToItsWest`, `hasToItsSoutheast`... or `isbnNumber`, `issnNumber` , `LCCN_Number` (sic) or `oclcNumber` although this is the kind of regularity that a learning algorithm should be able to take advantage of.

One of the goals of this data compilation was to develop enough training data for the development of learning algorithms, so it would have been convenient to merge some predicates in order to have more data for each. Following some of my suggestions, the organizers have already produced a second version of the data, merging many similarly named predicates. But there are still interesting cases left: `address` and `location` or `mission` and `crewMembers` (its inverse) are different predicates that could have been normalized. The same remark applied to `order` which is the inverse of `class` or `division`.

Contrarily, some predicates should be split because they are used in different contexts: for example, `language` can be used for the language spoken by a person or the language in which a book is written. This would not have appeared if the usual naming convention for predicates had been followed.

Of course, predicates being arbitrary URI, their *name* is not in principle important, but a system developed without caring for basic conventions would perhaps not be very appealing for Semantic Web enthusiasts.

Another suggestion to better reflect NLG challenges would be to split complex objects such as in the following `mtriple`s:

```
Arros_negre | mainIngredients | "White␣rice,␣cuttlefish␣or␣squid,␣
   cephalopod␣ink,␣cubanelle␣peppers"
Bacon_sandwich | alternativeName | "Bacon␣butty,␣bacon␣sarnie,␣rasher␣
```

```
    sandwich␣bacon␣sanger␣piece␣'n␣bacon␣bacon␣cob␣bacon␣barm␣bacon␣
    muffin"
Acharya_Institute_of_Technology | campus | "In␣Soldevanahalli␣Acharya␣Dr.
    ␣Sarvapalli␣Radhakrishnan␣Road␣Hessarghatta␣Main␣Road␣Bangalore␣
    560090."
Alvis_Speed_25 | transmission | "single␣plate␣clutch␣separate␣4-speed␣
    gearbox␣all-silent␣and␣all-syncromesh␣centre␣change␣lever␣open␣
    tubular␣propellor␣shaft␣with␣metal␣joints␣␣spiral␣bevel␣fully␣floating
    ␣back␣axle"
```

Each of these triples should be split into many each with an object containing a single ingredient or name. Ideally, it would be an RDF collection of single ingredients or names. Currently, simply copying the object probably artificially inflates the similarity scores since these enumerations have many words in common with the references.

The task is still far from trivial but, given these caveats, we think that developers should not extrapolate too much on how the performance of their system on the WebNLG Challenge 2020 dataset could be replicated in *real-life* RDF contexts. Moreover RDF triples are usually linked with an ontology whose content should be integrated into the realizer, a context that is not taken into account in WebNLG Challenge 2020.

7 Conclusion

This paper has described a symbolic approach to solve the shared task WebNLG Challenge 2020. The automatic scores are quite competitive compared to the ones of the other participants which, we conjecture, most often used machine learning approaches. The system is very fast on a standard laptop. It can also be quite easy to adapt to a new domain: considering about one minute per new predicate, it would have taken about 3 hours to develop 200 new predicates. It would be interesting if machine learning could be applied for developing new templates, even though we doubt it would be as fast.

8 Acknowledgments

We thank our colleagues Fabrizo Gotti and Philippe Langlais for constructive comments on a first version of this paper.

References

Aguado, G., A. Bañón, J. Bateman, M. Bernardos Galindo, M. Fernández-López, A. Gomez-Perez, E. Nieto, A. Olalla, R. Plaza, and A. Sánchez
1998. ONTOGENERATION: Reusing domain and linguistic ontologies for spanish text generation. In *Proceedings of the 13th European Conference on Artificial Intelligence (ECAI'98)*, Brighton, UK.

Bateman, J.
1997. Enabling technology for multilingual natural language generation: the KPML development environment. *Natural Language Engineering*, 3(1):15–55.

Bateman, J. A., R. Henschel, and F. Rinaldi
1995. Generalized Upper Model 2.0: documentation. Technical report, GMD/Institut für Integrierte Publikations- und Informationssysteme, Darmstadt, Germany.

Beckett, D., T. Berners-Lee, E. Prud'hommeaux, and G. Carothers
2014. RDF 1.1 Turtle - Terse RDF Triple Language. Technical report, W3C.

Bontcheva, K. and Y. Wilks
2004. Automatic report generation from ontologies: The MIAKT approach. In *Natural Language Processing and Information Systems*, F. Meziane and E. Métais, eds., Pp. 324–335, Berlin, Heidelberg. Springer Berlin Heidelberg.

Bouayad-Agha, N., G. Casamayor, and L. Wanner
2014. Natural language generation in the context of the semantic web. *Semantic Web*, 5:493–513.

Castro-Ferreira, T., C. Gardent, N. Ilinykh, C. van der Lee, S. Mille, D. Moussalem, and A. Shimorina
2020. 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). Association for Computational Linguistics.

Dong, N. T. and L. B. Holder
2014. Natural language generation from graphs. *International Journal of Semantic Computing*, 08(03):335–384.

Duboue, P. A. and K. R. McKeown
2003. Statistical acquisition of content selection rules for natural language generation. In *Proceedings of the 2003 Conference on Empirical Methods in Natural Language Processing*, Pp. 121–128.

Duma, D. and E. Klein
2013. Generating natural language from linked data: Unsupervised template extraction. In *Proceedings of the 10th International Conference on Computational Semantics (IWCS 2013) – Long Papers*, Pp. 83–94, Potsdam, Germany. Association for Computational Linguistics.

Ell, B. and A. Harth
2014. A language-independent method for the extraction of RDF verbalization templates. In *Proceedings of the 8th International Natural Language Generation Conference (INLG)*, Pp. 26–34, Philadelphia, Pennsylvania, U.S.A. Association for Computational Linguistics.

Gagnon, M. and L. Da Sylva
2006. Text compression by syntactic pruning. In *Proceedings of the 19th International Conference on Advances in Artificial Intelligence: Canadian Society for Computational Studies of Intelligence*, AI'06, Pp. 312–323, Berlin, Heidelberg. Springer-Verlag.

Galanis, D., G. Karakatsiotis, G. Lampouras, and I. Androutsopoulos
2009. An open-source natural language generator for OWL ontologies and its use in Protégé and Second Life. In *Proceedings of the Demonstrations Session at EACL 2009*, Pp. 17–20, Athens, Greece. Association for Computational Linguistics.

Gardent, C., A. Shimorina, S. Narayan, and L. Perez-Beltrachini
2017a. Creating training corpora for NLG micro-planners. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Pp. 179–188, Vancouver, Canada. Association for Computational Linguistics.

Gardent, C., A. Shimorina, S. Narayan, and L. Perez-Beltrachini
2017b. The WebNLG challenge: Generating text from RDF data. In *Proceedings of the 10th International Conference on Natural Language Generation*, Pp. 124–133, Santiago de Compostela, Spain. Association for Computational Linguistics.

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

Molins, P. and G. Lapalme
2015. JSrealB: A bilingual text realizer for web programming. In *Proceedings of the 15th European Workshop on Natural Language Generation (ENLG)*, Pp. 109–111, Brighton, UK. Association for Computational Linguistics.

Perez-Beltrachini, L., R. Sayed, and C. Gardent
2016. Building RDF content for data-to-text generation. In *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, Pp. 1493–1502, Osaka, Japan. The COLING 2016 Organizing Committee.

Popović, M.
2015. chrF: character n-gram F-score for automatic MT evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, Pp. 392–395, Lisbon, Portugal. Association for Computational Linguistics.

Sun, X. and C. Mellish
2006. Domain independent sentence generation from RDF representations for the Semantic Web. In *ECAI06 Combined Workshop on Language-Enabled Educational Technology and Development and Evaluation of Robust Spoken Dialogue Systems*.

Uschold, M.
2018. Demystifying OWL for the enterprise. *Synthesis Lectures on the Semantic Web: Theory and Technology*, 8(1):i–237.

Vougiouklis, P., H. Elsahar, L.-A. Kaffee, C. Gravier, F. Laforest, J. Hare, and E. Simperl
2018. Neural Wikipedian: Generating textual summaries from knowledge base triples. *Journal of Web Semantics*, 52-53:1 – 15.

Wilcock, G. and K. Jokinen
2003. Generating responses and explanations from RDF/XML and DAML+OIL. In *Knowledge and Reasoning in Practical Dialogue Systems*, Pp. 58–63. Volume: Proceeding volume:.

Zhu, Y., J. Wan, Z. Zhou, L. Chen, L. Qiu, W. Zhang, X. Jiang, and Y. Yu
2019. Triple-to-Text: Converting RDF triples into high-quality natural languages via optimizing an inverse KL divergence. In *Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR'19, Pp. 455–464, New York, NY, USA. Association for Computing Machinery.

Appendix

Triples of the top part of Table 1 rendered as RDF-XML.

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:dbo="http://dbpedia.org/ontology/"
    xmlns:dbr="http://dbpedia.org/resource/"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:dbp="http://dbpedia.org/property/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
  <dbo:Astronaut rdf:about="http://dbpedia.org/resource/Alan_Shepard">
    <rdfs:label>Alan Shepard@en</rdfs:label>
    <dbo:birthDate rdf:datatype="http://www.w3.org/2001/XMLSchema#date"
    >1923-11-18</dbo:birthDate>
    <dbo:birthPlace>
      <rdf:Description rdf:about="http://dbpedia.org/resource/
         New_Hampshire">
        <rdfs:label xml:lang="en">New Hampshire</rdfs:label>
      </rdf:Description>
    </dbo:birthPlace>
    <dbo:mission>
      <rdf:Description rdf:about="http://dbpedia.org/resource/Apollo_14">
        <dbp:operator>
          <rdf:Description rdf:about="http://dbpedia.org/resource/NASA">
            <rdfs:label xml:lang="en">NASA</rdfs:label>
          </rdf:Description>
        </dbp:operator>
        <rdfs:label xml:lang="en">Apollo 14</rdfs:label>
      </rdf:Description>
    </dbo:mission>
    <dbp:dateOfRet rdf:datatype="http://www.w3.org/2001/XMLSchema#date"
    >1974-08-01</dbp:dateOfRet>
    <dbo:deathPlace>
      <rdf:Description rdf:about="http://dbpedia.org/resource/California">
        <rdfs:label xml:lang="en">California</rdfs:label>
      </rdf:Description>
    </dbo:deathPlace>
  </dbo:Astronaut>
</rdf:RDF>
```

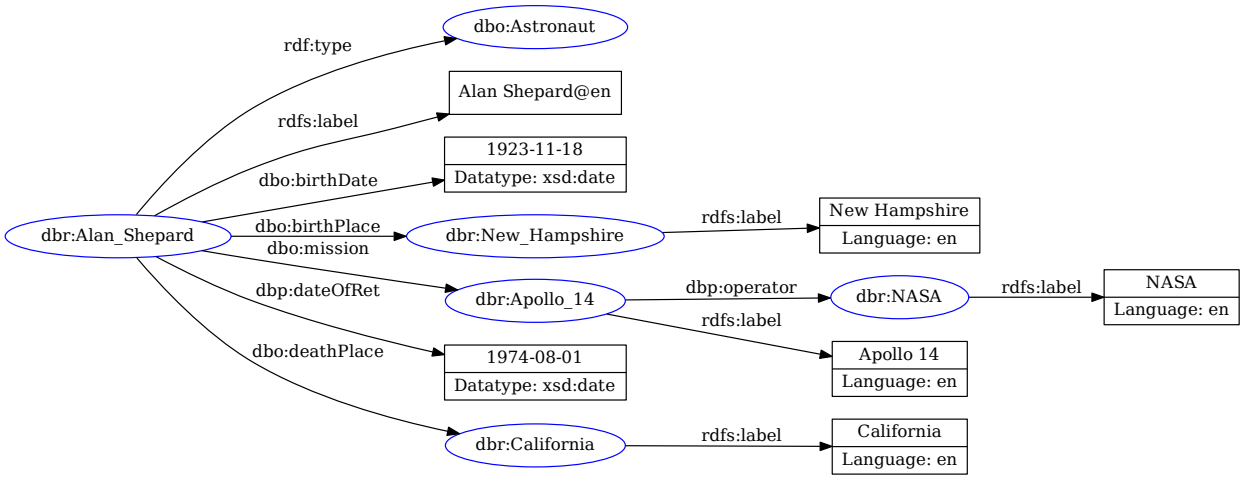

Figure 1: Visualization of the graph of the triples of the top of Table 1. This graph was created with `http://www.ldf.fi/service/rdf-grapher` from the Turtle version of these triples.

Triples of the top part of Table 1 rendered as Turtle.

```
@prefix dbo:     <http://dbpedia.org/ontology/> .
@prefix dbp:     <http://dbpedia.org/property/> .
@prefix dbr:     <http://dbpedia.org/resource/> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd:     <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs:    <http://www.w3.org/2000/01/rdf-schema#> .

dbr:Alan_Shepard a dbo:Astronaut ;
      rdfs:label "Alan␣Shepard@en" ;
      dbo:birthDate "1923-11-18"^^xsd:date ;
      dbo:birthPlace dbr:New_Hampshire ;
      dbo:deathPlace dbr:California ;
      dbo:mission dbr:Apollo_14 ;
      dbp:dateOfRet "1974-08-01"^^xsd:date .

dbr:New_Hampshire rdfs:label "New␣Hampshire"@en .

dbr:California rdfs:label "California"@en .

dbr:Apollo_14 rdfs:label "Apollo␣14"@en ;
      dbp:operator dbr:NASA .

dbr:NASA rdfs:label "NASA"@en .
```

Baseline

```
def camel_case_split(s):
    return list(map(str.lower,
    re.findall(r'([A-Z0-9]+|[A-Z0-9]?[a-z0-9]+)(?=[A-Z0-9]|\b)', s))
       )

def realizeTriple(triple):
    return cleanNode(triple.s)+"␣"+\
      "␣".join(map(str.lower,camel_case_split(triple.p)))\
      +"␣"+cleanNode(triple.o)

def cleanNode(n):
    return n.replace("_language","").replace("_","␣").replace('"','')
       )
```