Highlights

Better than XML: towards a lexicographic markup language

Michal Měchura

- Dictionaries encoded in XML are unnecessarily verbose and complex due to overuse of purely structural markup.
- Much data in lexicography is inherently *headed*, but headedness is difficult to represent in XML.
- JSON and YAML are no better than XML at representing headedness and at serving the needs of lexicography.
- Two existing languages do provide good support for headed data: XML's historical predecessor SGML and a less well-known language called NVH.

Better than XML: towards a lexicographic markup language

Michal Měchura[a]

[a]*Faculty of Informatics, Masaryk University, Brno, Czech Republic*

Abstract

This article takes a critical look at how XML is used in lexicography and asks the question, why do dictionary entries often end up looking so complex when encoded in XML? The main reason for the perceived complexity of XML-encoded dictionaries is *purely structural markup*: XML elements which contain other XML elements instead of human-readable text. The over-abundance of purely structural markup in lexicography is caused by the nature of lexicographic content, much of which is inherently *headed*. XML has no support for headedness and neither do other commonly used languages such as JSON and YAML. In this article we propose a number of constraints and extensions to XML, JSON and YAML which add support for headedness into these languages.

Keywords: XML, JSON, YAML, Verbose markup, Data models, Lexicography

1. Introduction: dictionaries and XML

Lexicography is the discipline of creating dictionaries (where by dictionaries we mean books, websites and apps where human users look up information about words). In modern lexicography, dictionary entries are usually encoded in XML [1]. Each dictionary entry is typically its own XML document, and each such XML document conforms to an XML schema created for that particular dictionary. An example can be seen in listing 1 below which shows how a dictionary entry from a bilingual dictionary would typically be encoded in XML. For comparison, figure 1 shows how the same entry would eventually be presented to a human user.

absolutely *adv*

1. *(completely)* go hiomlán, go huile agus go hiomlán
   I absolutely agree *aontaím go huile agus go hiomlán*
2. *(very)* amach is amach, ar fad
   he's absolutely brilliant *tá sé ar fheabhas amach is amach*

Figure 1: A human-readable rendering of a dictionary entry encoded in XML

Listing 1: A dictionary entry encoded in XML

```
<entry>
    <headword>absolutely</headword>
    <pos>adv</pos>
    <sense>
       <gloss>completely</gloss>
       <translation>go hiomlán</translation>
       <translation>go huile agus go hiomlán</translation>
       <exampleContainer>
            <example>I absolutely agree</example>
            <translation>aontaím go huile agus go hiomlán</translation>
         </exampleContainer>
    </sense>
    <sense>
       <gloss>very</gloss>
       <translation>amach is amach</translation>
       <translation>ar fad</translation>
       <exampleContainer>
            <example>he's absolutely brilliant</example>
            <translation>tá sé ar fheabhas amach is amach</translation>
       </exampleContainer>
    </sense>
</entry>
```

Notice that the XML encoding is relatively high-level: it encodes the *structure* of the entry, not its *appearance* on screen or on paper. There are XML elements to indicate where the headword is, where one sense ends and another begins, and so on. So, we can define *dictionary encoding* as the activity of taking an inventory of lexicographically relevent content items such as headwords, part-of-speech labels, senses and translations, and expressing them formally in a language such as XML.

XML is the most commonly used encoding language for dictionaries today. As lexicography began digitising itself in the late 1990s and early 2000s,

XML seemed like an obvious choice: for example, an early seminal paper on dictionary encoding [2] extols the virtues of XML and does not even consider any alternatives – to be sure, no workable alternatives to XML existed in the early years of digital lexicography. XML was already popular for text encoding in general, and its underlying tree-like object model fitted in nicely with pre-existing thinking in theoretical lexicography where dictionary entries were modelled as tree structures [3].

It is the 2020s now and lexicography has long transitioned from paper to screens. The focus has moved from *retrodigitising* old paper-bound dictionaries to producing new *born-digital* ones. There have been advances in automation, so that we no longer talk of *writing* dictionaries but *generating* them from data and then *post-editing* them [4]. There have been quantitative advances in both scale (how many dictionaries are produced, how large they are) and speed (how quickly). XML is still with us in this new world.

This article asks whether XML is still fit for the job. Some of the recent advances in digital lexicography have given rise to scenarios and use cases which were not there in the early years, such as the need to change dictionary schemas frequently during the lifetime of a project, or to make dictionaries more easily processable by machines (as opposed to merely legible to humans). The purpose of this article is to show that XML makes some of these tasks unnecessarily difficult, and to look for alternatives.

2. The dark side of XML in lexicography

XML has many properties wich make it a good language for encoding dictionary entries, for example the fact that XML preserves the order of elements, or that XML has out-of-the-box support for inline markup. Later in this article (in section 7) we will give a detailed analysis of those features of XML which are good for dictionary encoding. At this point, however, we are going to concentrate on occasions when the use of XML in lexicography is more hindrance than help.

Since its emergence in the late 1990s[1] and despite its popularity, XML has been subject to passionate criticism from many quarters [5]. The usual objection is that XML is a "verbose" language, which is another way of saying

[1]The W3C XML recommendation, the de-facto standard for XML, was published in 1998.

that XML documents tend to have an inconveniently high ratio of tags to content: it takes a lot of tags to encode a little content.

Some of the perceived verbosity of XML is caused by superficial design decisions in the syntax of XML, in particular the fact that the name of each element needs to be given twice, first in the opening tag and then again in the closing tag, which is obviously redundant. This is, however, not the only reason why XML looks and feels verbose. There are other, less superficial reasons for the perceived verbosity of XML, reasons which have less to do with the syntax and more with the underlying data model. Nowhere is this more apparent than in lexicography, as we will show in the rest of this section.

2.1. Purely structural markup and matryoshkization

We will concentrate here on one less obvious cause of verbosity in XML: the multi-layered embedding of elements inside other elements inside yet more elements, a phenomenon we call *matryoshkization*.[2] Listing 2, which shows how a pair of translations would typically be encoded somewhere inside a bilingual dictionary, demonstrates matryoshkization in practice.

Listing 2: A pair of translations encoded in XML

```
<translationGroup>
  <translationContainer>
    <translation>leasú</translation>
    <pos>n-masc</pos>
  </translationContainer>
  <translationContainer>
    <translation>athchóiriú</translation>
    <pos>n-masc</pos>
  </translationContainer>
</translationGroup>
```

The only XML elements here that contain actual human-readable content are `<translation>` (= the translation's wording) and `<pos>` (= its part of speech). The remaining XML elements are purely structural, used for grouping other elements together:

- The `<translationContainer>` element groups `<translation>` and `<pos>` elements together.

[2] A *matryoshka* is a popular Russian wooden toy in the form of a doll. When the doll is opened it reveals a smaller doll inside, which in turn has another smaller doll inside, and so on.

- The `<translationGroup>` element groups several `<translationContainer>` elements together.

Let us walk ourselves through the hypothetical steps which may have led a schema designer to designing the schema in this way.

Step 1. In the beginning, the requirement was to encode translations. This can be done very easily with just one type of element which we can call `<translation>`: listing 3.

Listing 3: Two translations

```
<translation>leasú</translation>
<translation>athchóiriú</translation>
```

Step 2. Then the schema designer realised that we need to encode part-of-speech labels for each translation, using an element we can call `<pos>`: listing 4.

Listing 4: Two translations and two POS labels

```
<translation>leasú</translation>
<pos>n-masc</pos>
<translation>athchóiriú</translation>
<pos>n-masc</pos>
```

Step 3. But, to indicate which part-of-speech element belongs to which translation, the schema designer decides to group each pair under a common parent. A popular naming convention in lexicography is to call such elements *containers*, for example `<translationContainer>`: see listing 5. This has introduced one level of *matryoshkization* into the entry schema: one layer of purely structural markup.

Listing 5: One layer of purely structural markup

```
<translationContainer>
    <translation>leasú</translation>
    <pos>n-masc</pos>
</translationContainer>
<translationContainer>
    <translation>athchóiriú</translation>
    <pos>n-masc</pos>
</translationContainer>
```

Step 4. At this point, the schema designer notices that the code in which translations are encoded is quite long. It occurs to him or her that it might be a good idea to wrap all translation containers inside yet another layer of

purely structural markup, so that it becomes easier to collapse and expand in an XML editor. In lexicography, when an element's only purpose is to group a list of elements of the same type together, a popular naming convention is to call it a *group*, for example `<translationGroup>`: listing 6.

Listing 6: Two layers of purely structural markup

```
<translationGroup>
  <translationContainer>
    <translation>leasú</translation>
    <pos>n-masc</pos>
  </translationContainer>
  <translationContainer>
    <translation>athchóiriú</translation>
    <pos>n-masc</pos>
  </translationContainer>
</translationGroup>
```

We have ended up with two layers of purely structural markup in the entry schema. The source code of our entries has become difficult for humans to read and navigate while editing. Most of the tags are purely structural, while tags which surround actual human-readable content are the minority.

The trouble is, however, that the purely structural markup is *not* redundant. It (or most of it) is there to encode lexicographically relevant facts, such as the fact that this part-of-speech label belongs to this translation. The matryoshkization seems unavoidable, a necessary consequence if one wants to encode the facts one wants to encode. In the author's experience, lexicographers (and more importantly, IT professionals working in lexicography) often tacitly accept highly verbose XML as a necessary evil, as an inconvenience which needs to be accepted because there is no other way.

2.2. Matryoshkization versus your entry editor

A frequent counter-objection is that matryoshkization is not a problem because editing tools can hide the verbosity from the human lexicographer. It is, of course, possible in principle to create editorial user interfaces which do not expose the human lexicographer to the verbosity of the underlying XML. In practice, however, this is almost never done. All dictionary writing systems in wide use today, including Lexonomy [6], TLex [7], iLex [8] and the IDM Entry Editor [9], are basically schema-driven XML editors where the lexicographer is fully exposed to the verbosity of the purely structural markup (example in figure 2).

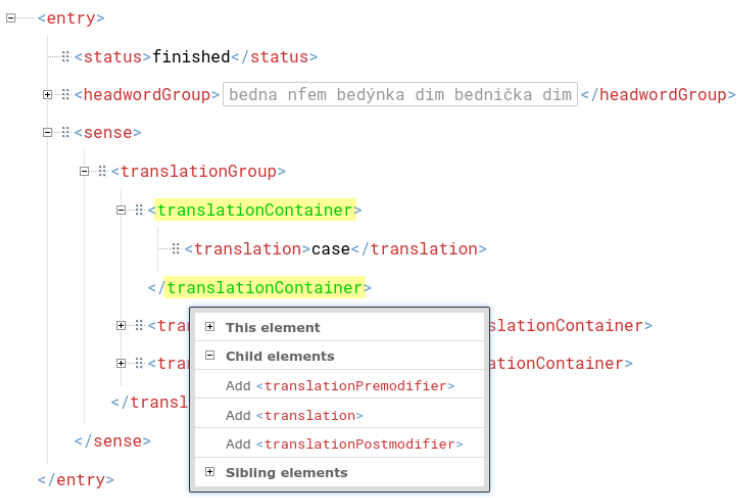

Figure 2: A typical lexicographic XML editor (Lexonomy)

To "hide" the XML from the lexicographer, one needs to develop a customised editorial UI which is specific to that particular dictionary (or, more accurately, to that particular entry schema). This can be a non-trivial software development task, especially if one considers the necessity to maintain the UI throughout the lifetime of the project and to keep it synchronised with changes to the schema. Most dictionary projects do not have the staff or the budget for such software development effort. Most dictionary projects simply procure an off-the-shelf dictionary writing system and customise it with their own entry schema. Dictionary writing systems typically do not even allow much more customisation than that. The only widely used dictionary writing system where the XML *can* be hidden behind a custom-built entry editing "widget" is Lexonomy, but this feature is rarely used there – precisely for the reason that developing and maintaining the widgets is expensive.

Therefore, it is invalid to claim that matryoshkization does not matter because it can be hidden. Matryoshkization cannot easily be – and rarely is – hidden from human lexicographers. Matryoshkization is a real and existing inconvenience on many dictionary projects.

2.3. Matryoshkization versus schema migration

The fact that entries are difficult to read and navigate for human lexicographers is not the only consequence of matryoshkization. Another con-

sequence is that almost every change to the entry schema renders existing entries invalid.

Let us illustrate that by returning to the hypothetical example of a schema designer who is in the process of designing an entry schema for a new dictionary project. In **step 1**, the designer has designed a schema which allows translations to be encoded in the simplest possible way, using just one type of element called `<translation>`: listing 7.

Listing 7: Translations without POS labels

```
<translation>leasú</translation>
<translation>athchóiriú</translation>
```

The project starts and several hundreds of entries are encoded using this schema. Then the requirements change and it transpires that we need to add part-of-speech labels to some (but not all) translations. The schema designer goes back to the drawing board and follows through with **steps 2 and 3**: the schema is changed so that translations are now to be encoded in a `<transationContainer>` element which can have two child elements, one `<translation>` and zero or more `<pos>`: listing 8.

Listing 8: Translations with optional POS labels

```
<translationContainer>
    <translation>leasú</translation>
</translationContainer>
<translationContainer>
    <translation>athchóiriú</translation>
    <pos>n-masc</pos>
</translationContainer>
```

The schema designer has made two changes into the schema: (1) the new `<pos>` element type is now an optional sibling of `<translation>` and (2) the new `<translationContainer>` element type has taken the place of `<translation>`, "demoting" it to the role of its child. The first change does not cause pre-existing entries to be invalid, but the second one does. The consequence is that all previously encoded entries are now invalid as per the new schema – including, frustratingly, those entries where we are *not* planning to add any part-of-speech labels. So, changing the schema was only half the work: we also need to write a *schema migration script* to make the existing entries valid again.

Every time we add new *non-structural* element types into an entry schema, such as `<pos>` in our example, the change usually does not cause pre-existing

entries to be invalid (as long as the new element is optional). But when we add new *purely structural markup* into the schema, like we did when we introduced `<translationContainer>`, the schema becomes matryoshkized, all pre-existing entries become invalid and we need to fix that with a schema migration script. In other words, matryoshkization not only makes entries verbose, it also makes schema migrations more difficult.

A possible counter-objection is that this (= the necessity to write schema migration scripts every time we matryoshkize the schema) is unavoidable because the matryoshkization itself is unavoidable: there is no other way to encode what we want to encode than through purely structural markup. To be sure, this inconvenience is not unique to XML, schema migration scripts are common everywhere data is managed, in particular in relational databases. But that is beside the point. Avoidable or unavoidable, matryoshkization (and the necessity for schema migration scripts) is a hindrance to *agility* in the dictionary building process: it prevents the schema designer from making changes to the schema unreluctantly and frequently, in response to evolving project requirements.

2.4. Look-ahead matryoshkization

Experienced schema designers are often keen to avoid having to change the entry schema halfway through a project. For that reason, schema designers often choose to matryoshkize the schema even if there is no need for it *yet*, a phenomenon we can call *look-ahead matryoshkization*. For example, when designing a schema for encoding translations, the designer may introduce the purely structural element `<translationContainer>` from the very start, as in listing 9, even though there is no need for it and a `<translation>` element on its own would do. The designer is hoping to future-proof his or her schema: should a requirement for a `<pos>` sibling to `<translation>` emerge in the future, he or she will be able to introduce it into the schema without invalidating existing entries and without having to write a schema-migration script. This is perhaps wise and prudent – but if that requirement never emerges, then we have ended up with a dictionary full of XML-encoded entries which are more verbose than they need to be.

Listing 9: Translations with look-ahead matryoshkization

```
<translationContainer>
    <translation>leasú</translation>
</translationContainer>
<translationContainer>
```

```
    <translation>athchóiriú</translation>
</translationContainer>
```

2.5. Summary: XML in lexicography

Dictionary entries, when encoded in XML, tend to be overly verbose due to a phenomenon called *matryoshkization*. Matryoshkization is caused by the presence of purely structural markup. In addition to verbosity, matryoshkization also causes difficulties during schema updates.

Some degree of matryoshkization and purely structural markup can be observed in practically every discipline where XML is used, but (arguably) it is more prevalent in lexicography than anywhere else. So, in the next two sections, we are going to analyse in more detail the patterns of purely structural markup which occur often in lexicography and we will ask the question, what is so special about lexicographic data that makes matryoshkization so prevalent?

3. Patterns of purely structural markup

We can define purely structural markup as such XML elements which contain no text nodes as their direct children: all their child nodes are other XML elements. We have seen how too much structural markup leads to the phenomenon of *matryoshkization*, which is a special subcase of the phenomenon of *verbosity* for which XML is often criticised. Let us now review the patterns of purely structural markup that commonly occur in lexicography. Broadly speaking, there are two patterns: the 'list' pattern and the 'headed' pattern.

3.1. The 'list' pattern of purely structural markup

Listing 10: Example of the 'list' pattern

```
<translations>
  <translationContainer>...</translationContainer>
  <translationContainer>...</translationContainer>
  <translationContainer>...</translationContainer>
</translations>
```

The first pattern is where a parent element wraps a sequence of child elements which are all of the same type. It is there because the designer of the schema probably thought it useful to group elements of the same type

under a common parent element, like in **Step 4** of our fictional but realistic schema design process.

The usefulness of this grouping is debatable. The group thus created does not seem to represent any lexicographic fact which a lexicographer might want to communicate to the dictionary's end-users. The parent wrapper is almost always unnecessary in the sense that it conveys no information which could not be inferred: the fact that there exists a list of translations is obvious from the fact that there is a sequence of `<translation>` elements in the entry. Grouping them under a common parent does not contribute any new information.

Unnecessary grouping of this kind can be found in XML outside lexicography too and tends to be advised against in XML styleguides [10]. The 'list' pattern can almost always be explained away as a bad practice, and the dictionary schema can be made less complex by simply removing the purely structural elements.

3.1.1. The 'headed' pattern of purely structural markup

Listing 11: Example of the 'headed' pattern

```
<translationContainer>
  <translation>athchóiriú</translation>
  <pos>n-masc</pos>
  <usage>formal</usage>
</translationContainer>
```

The second pattern is where a parent element wraps child elements of different types, one of which can be considered the "head" and the others can be seen as providing additional information about the head. An example is `<translationContainer>` which can be said to be headed by `<translation>`, while the other children `<pos>` and `<usage>` provide additional information about the head.

Unlike the list pattern, the headed pattern cannot be explained away as a bad practice. Its purpose is to encode lexicographic facts which the lexicographer wants to communicate to the end-user, for example the fact that this `<pos>` element belongs to this `<translation>` element. The purely structural `<translationContainer>` element is a tool for encoding that fact.

Whenever during the process of designing an entry schema for a dictionary a requirement arises to encode something which appears to have a "head" plus a few other elements that provide additional information about the head,

the headed pattern of purely structural markup is a popular choice – as it was for our fictional schema designer in **Step 3** above.

Why is the headed pattern of purely structural markup so popular in lexicography? The reason is that much of lexicographic content inherently *is* headed: we will show multiple examples of that in the following section.

4. The headedness of lexicographic data

In XML, at an abstract level, every XML element can be seen as a pair of two things: a name and a value. The **name** is what we have in the opening and closing tags, while everything between the tags is the **value** which can be either plain text, or a list of child elements, or a mixture of both (so-called mixed content), or it can be empty. But the point is that an XML element always consists of exactly two things: a name and a value, even if the value is complex.

In lexicography, on the other hand, much of the content we encounter could more efficiently be modelled as a *triple*, as a group of three things: a name, a value, and a list of modifiers containing zero, one or more other such triples. The name and the value together are the *head*. Many content objects in lexicography are inherently headed, but headedness is difficult to model in XML without purely structural markup. Let us look at some examples of lexicographic content objects which are headed.

4.1. Translations are headed structures

Listing 12: A typical XML encoding of a translation

```
<translationContainer>
  <translation>athchóiriú</translation>
  <pos>n-masc</pos>
  <usage>formal</usage>
<translationContainer>
```

Listing 13: The same translation in concise pseudocode

```
translation: athchóiriú
  pos: n-masc
  usage: formal
```

In many bilingual dictionaries, translations are given simply as strings of text with no other information. Such translations are not headed, of course. But, in an encoding-oriented dictionary (ie. a dictionary which

tells you how to express something in a language in which you are not fluent), translations are often decorated with grammatical annotations (part-of-speech labels) and pragmatic annotations (usage labels). Such translations *are* headed: the `<translation>` element together with its plain-text value is the head, while the other elements (`<pos>` and `<usage>`) are modifiers of the head. Purely structural markup (in the form of a parent element such as `<translationContainer>`) is often used to encode this in XML.

4.2. Example sentences are headed structures

Listing 14: A typical XML encoding of an example sentence

```
<exampleContainer>
  <example>Ich nehme den Regenschirm mit.</example>
  <source>bib-147_12</source>
  <translation>I'll take my umbrella with me.</translation>
</exampleContainer>
```

Listing 15: The same example sentence in concise pseudocode

```
example: Ich nehme den Regenschirm mit.
  source: bib-147_12
  translation: I'll take my umbrella with me.
```

In many dictionaries, example sentences are not just strings of text: they come with additional content such as bibliographical references (to tell us where the example comes from), usage labels (to tell us, for instance, that this sentence is colloquial) and translations. In other words, dictionary examples are headed structures: the `<example>` element together with its plain-text value is the head, while the other elements are modifiers of the head. Some of the modifier elements can be headed structures too: for instance, it is imaginable that translations could have their own modifiers, as in listing 16.

Listing 16: Example sentences have translations which have usage labels

```
example: Ich gehe auf Nummer sicher.
  translation: I'll play it safe.
    usage: informal
  translation: I'll stay on the safe side.
    usage: neutral
  translation: I will err on the side of caution.
    usage: formal
```

4.3. Collocations are headed structures

Listing 17: A typical XML encoding of a collocate

```
<collocation>
  <collocate>make</collocate>
  <example>I have made a mistake.</example>
  <example>Everybody makes mistakes.</example>
</collocation>
```

Listing 18: The same collocate in concise pseudocode

```
collocate: make
  example: I have made a mistake.
  example: Everybody makes mistakes.
```

It is becoming common for dictionaries to contain information about the *collocates* of the headword: words which often occur together with the headword in real-world language use. For instance, inside the entry for the headword 'mistake' we might find a block of information that tells us that the headword collocates with the verb 'make' (as in 'to make a mistake'), and then gives us some additional information about this collocation, such as some usage labels or a few example sentences. So, in a dictionary entry, collocations are headed structures: the `<collocate>` element together with its plain-text value is the head, while the other elements are modifiers of the head.

4.4. Senses can be headed structures too

Listing 19: A typical XML encoding of a sense

```
<sense>
  <definition>an institution where you store money</definition>
  <translation>banque</translation>
  <example>I got a large loan from the bank.</example>
</sense>
```

Listing 20: The same sense in concise pseudocode

```
definition: an institution where you store money
  translation: banque
  example: I got a large loan from the bank.
```

In lexicography, a dictionary entry is typically subdivided into one or more senses. A sense is a container for things such as definitions, translations and examples. Normally, a sense is *not* a headed structure because there is no

obvious "head": no single element inside the sense where we could say that all other elements are its modifiers. In XML, senses are practically always encoded by means of purely structural markup: there is a `<sense>` element which has no plain-text children of its own, but has many child elements such as `<definition>`, `<translation>` and `<example>`.

But is it true that senses are not headed structures? There is a case to be made that *definitions* are the heads of senses. A definition says that such-and-such meaning of the headword exists, and the remaining elements inside the sense can be understood as providing additional information about that meaning.

Not all dictionaries contain definitions. But, in those those that do, it is possible to understand senses as headed structures. In an XML encoding of senses, the `<sense>` element is yet another incarnation of the 'headed' pattern of structural markup.

4.5. Entries can be headed structures too

Listing 21: A typical XML encoding of an entry

```
<entry>
  <headword>bank<headword>
  <partOfSpeech>noun</partOfSpeech>
  <sense>an institution where...</sense>
  <sense>a stretch of land...</sense>
</entry>
```

Listing 22: The same entry in concise pseudocode

```
headword: bank
  partOfSpeech: noun
  sense: an institution where...
  sense: a stretch of land...
```

We can perform the same re-analysis on entries as we did on senses. Entries do not seem like obviously headed structures: they are simply containers for various elements such as headwords, part-of-speech labels and senses. But one of then does stand as a possible candidate for being the head: the headword! It is, after all, called a headword for one good reason: its purpose is to *head* the entire entry, while the rest of the entry is *about* the headword. On that analysis, even entire dictionary entries can be understood as headed structures, and the very existence of an `<entry>` element in XML-encoded dictionaries can be understood as an incarnation of the 'headed' pattern of structural markup, a consequence of matryoshkization.

5. How to encode headedness in XML

We have seen in the previous section that headed content structures are far from uncommon in lexicography: it so happens that much of dictionary content is inherently headed. And, in the sections before that, we have seen that to encode headed structures in XML, purely structural markup (more specifically, the 'headed' pattern of purely structural markup) is commonly used in lexicography, and that this is problematic because it has negative implications on readability and because it causes complications during schema updates.

The question to ask now is, are there other ways to encode headedness in XML? Is it possible to encode headed structures in XML without recourse to purely structural markup? In this section we will evaluate several options, some obvious and some less so.

5.1. Strategy 1: parentless sequencing

We have said before that the purpose of purely structural markup (in the 'headed' pattern) is to group elements together: to indicate which `<pos>` belongs to which `<translation>` and so on. Theoretically, it might be possible to achieve the same goal without purely structural markup, by relying only on the listing order of elements, as in listing 23.

Listing 23: Two headed structures encoded as parentless sequencing

```
<translation>leasú</translation>
<pos>n-masc</pos>
<translation>athchóiriú</translation>
<pos>n-masc</pos>
```

In this scenario, we would "know" that each `<pos>` element belongs to its nearest preceding sibling `<translation>` element. The problem with this approach is that this fact is not encoded explicitly in the XML, and tools processing this XML in the future may not "know" it as we "know" it now: to an XML parser, `<pos>` and `<translation>` are simply siblings and nothing else. We would need to program additional logic on top of the XML parser to make that explicit. So, parentless sequencing defeats the purpose of encoding entries in XML in the first place: to take facts which are implicit and make them explicit.

5.2. Strategy 2: mixed content

Yet another suggestion is to encode headedness as mixed content. Mixed content is a strategy used in XML to encode inline markup, a typical example is tags such as `<b>`, `<i>` and `<a href="...">` in HMTL: see listing 24.

Listing 24: HTML with mixed content

```
<p>
  This is <b>very</b> important.
</p>
```

To say that an XML element has "mixed content" is another way of saying that its child nodes are a sequence of text nodes and elements. This is a good strategy for encoding inline markup. Could it be a good strategy for encoding headedness, as in listing 25?

Listing 25: HTML with mixed content

```
<translation>
  athchóiriú
  <pos>n-masc</pos>
  <usage>formal</usage>
</translation>
```

The problem is that there is no formal distinction (to an XML parser) between the head (= the element's first child) and the modifiers (= the element's other children). If we ask an XML parser to give us the text of the `<translation>` element, it will give is a concatenation of all the text node descendants, which is the string `athchóiriú n-masc formal` (with whitespace collapsed).

The problem becomes more apparent if the head's value contains inline markup, like in listing 26. Here, the `<example>` element has four children: the text `to implement electoral ` (with a trailing space), followed by the `<h>` element, followed by two more elements. An XML parser has no way of knowing that the first two children are part of the head's value and the others are not.

Listing 26: A headed structure, encoded as mixed content, where the head has inline markup

```
<example>
  to implement electoral <h>reform</h>
  <source>EU legislation</source>
  <translation>leasú toghchánach a chur i bhfeidhm</translation>
</example>
```

The mixed content strategy is only one step away from purely structural markup. The one step is to take those children that constitute the head's value and wrap them in yet another element, as in listing 27.

Listing 27: XML with purely structural markup

```
<exampleContainer>
  <example>to implement electoral <h>reform</h></example>
  <source>EU legislation</source>
  <translation>leasú toghchánach a chur i bhfeidhm</translation>
</exampleContainer>
```

This is an improvement on the mixed content strategy because the head value is now explicitly demarcated from the rest. But the downside is that our schema is now matryoshkized, with all the disadvantages we have identified above.

5.3. Strategy 3: children as attributes

A simple suggestion that might occur to a schema designer wanting to avoid purely structural markup is to use XML attributes instead: the head would be encoded as an XML element and all its children would become its attributes, as in listing 28.

Listing 28: XML with children as attributes

```
<translation pos="n-masc" usage="formal">
  athchóiriú
</translation>
```

The problem with this suggestion is that it does not scale beyond a few simple examples. This is because XML attributes come with several inconvenient limitations:

- Attribute names have to be unique, meaning that there can never be, for example, two `pos` attributes or two `usage` attributes in an element.
- Attribute values are plain text with no structure. So, it is impossible for an attribute to have its own attributes, or any other kind of child nodes, or to contain a list of values. In other words, an XML attribute is similar to an XML element in that it is a name-value pair, but with the additional limitation that the value must be plain text.

5.4. Strategy 4: heads as attributes

he other way around to encode values as attributes with a pre-agreed name such as `value`. Children are then encoded as normal XML elements, as in listing 29.

Listing 29: XML with heads as attributes

```
<translation value="athchóiriú">
  <pos value="n-masc"/>
  <usage value="formal"/>
</translation>
```

This encodes headedness successfully but has an even larger problem than the previous strategy: now *all* values must be plain text, in-line markup is impossible everywhere.

5.5. Conclusion: headedness in XML

The conclusion for this section is that even though it is possible to find strategies in XML to avoid purely structural markup and/or to represent headedness, each strategy comes with its own trade-offs. These trade-offs may or may not be acceptable to the schema designer depending on the requirements of the project, for example whether in-line markup is needed or not.

6. How to encode headedness in other serialization languages

Lexicography abounds in headed structures but the markup language we use in lexicography most often, XML, was never designed for it and can only accommodate it awkwardly. This is unfortunate. But can we perhaps find another markup language to use in lexicography instead of XML, one that can encode headedness more gracefully? In this section we will evaluate JSON and YAML as currently popular alternatives to XML, we will also look at SGML as XML's historical predecessor, and we will also look at one less well-known language called NVH. In each case we will ask whether the language is able to encode headed structures without purely structural markup, and if not, how the language would need to change to support headedness.

6.1. Headedness in SGML

XML's historical predecessor was SGML [11, 12]. Invented primarily as a text markup language, SGML was[3] more complex than XML, but this complexity enabled many **markup minimisation features** which, in retrospect, made SGML into a language which supported headedness.

One of SGML's markup minimisation features was the ability to omit closing tags. Early versions of the HTML standard had a similar feature. So, it was possible to write code like in listing 30.

Listing 30: SGML with minimised markup

```
<translation>athchóiriú
<pos>n-masc
```

The parser would implicitly "assume" the missing closing tags from its knowledge of the document schema. If the schema says that the `<translation>` and `<pos>` elements can only have text content and no child elements, then obviously they must be siblings and the parser will read the code as if the closing tags where there, like in listing 31.

Listing 31: SGML without minimised markup

```
<translation>athchóiriú</translation>
<pos>n-masc</pos>
```

This features of SGML made it possible to write less verbose code, but it still does not turn SGML into a headedness-supporting language. The markup minimisation feature which does turn SGML into such a language is something called **implicit elements**. In SGML, is was possible the specify in the document schema that certain element tags can left out altogether, even though the parser would still "assume" them to be there. Let's demonstrate that on an example where we take a matryoshkized XML fragment and reencode it in SGML. We start with a fragment like in listing 32.

Listing 32: SGML with all elements explicit

```
<translation>
  <value>athchóiriú</value>
  <pos>n-masc</pos>
  <usage>formal</usage>
<translation>
```

[3]We are talking about SGML in the past tense, as if SGML no longer existed. This is of course not true, SGML is still exists. The past tense here is only a reflection of the fact that SGML is rarely used anymore, at least for new projects.

Then, in the document schema, we specify that the <value> element is implicit. It now becomes possible to leave its opening and closing tags out, as in listing 33.

Listing 33: SGML with an implicit element

```
<translation>
  athchóiriú
  <pos>n-masc</pos>
  <usage>formal</usage>
<translation>
```

This looks similar to our attempt to encode headedness in XML through mixed content, but the trick is that this is *not* mixed content. When parsing this code fragment, the SGML parser will understand from the schema that

1. <translation> is not allowed to have any text content, and
2. <translation> is required to have as its first child an element called <value> which is required to have text content.

These facts will trigger the SGML parser into interpreting the code as if the <value> element were actually there, like in the previous code sample. All this means that SGML could, in principle, be used in lexicography to encode headed structures in a such a way that schema migration does not cause problems. Let's assume we start with a simple entry schema where translations are encoded like in listing 34.

Listing 34: This SGML fragment validates in *both* schemas

```
<translation>
  athchóiriú
<translation>
```

If we then update the schema such that

1. <translation> is no longer allowed to contain text content, and
2. <translation> is required to a contain an implicit element called <value> (as well other optional children such as <pos> and <usage>)

then the original entries are still parsed as valid: the SGML parser "assumes" the implicit element to be there. We can matryoshkize the schema without having to matryoshkize the data, and no schema migration scripts are needed.

To the author's knowledge, however, this property of SGML was never taken advantage of in lexicography. Lexicography began digitising itself at

a time when SGML had already peaked in popularity and XML was seen as its successor. And, to be sure, the flexibility of SGML came at a cost, as SGML was computationally hard to implement: all the markup minimisation features made it difficult to write parsers for SGML. XML evolved out of SGML to solve precisely that problem, as a subset of SGML which is more easily processable by machines. In its evolution from SGML to XML, the language gained machine processability and became easy to adopt, but lost support for headedness and gained on verbosity.

6.2. Headedness in JSON

As a serialisation format for data, JSON [13, 14] is often claimed to be more easily human-readable than XML. JSON is definitely less verbose than XML, mainly because the names of objects do not have to be repeated at the end of every object, which makes JSON significantly faster for (uncompressed) transmission than XML [15]. Listing 35 shows how an entry fragment might be encoded in JSON.

Listing 35: How a translation might be encoded in JSON

```
{
  "translationContainer": {
    "translation: "athchóiriú",
    "pos": "n-masc",
    "usage": "formal"
  }
}
```

Apart from this, however, JSON has the same problem as XML: it does not support headed structures. The code in listing 35 is JSON's equivalent of matryoshkization and purely structural markup: `translationContainer` is the purely structural element because it is an object which contains no literal text as its immediate child, all its children are other objects.

None of the strategies discussed for XML in section 5 have equivalents in JSON. The parentless sequencing strategy is impossible in JSON because JSON requires the names inside an object to be unique: listing 36 is illegal in JSON. The mixed content stragegy is not an option either because JSON does not allow mixing literal values with name-value pairs: listing 37 is also illegal in JSON. The only way to represent mixed content in JSON is to use array syntax `[...]` which comes with its own share of purely structural markup. And finally, the remaining two options discussed for XML which

make use of attributes have no equivalents in JSON because there is no such thing as attributes in JSON.

Listing 36: Parentless sequencing (illegal in JSON)

```
{
    "translation": "leasú",
    "pos": "n-masc",
    "translation": "athchóiriú",
    "pos": "n-masc"
}
```

Listing 37: Mixed content (illegal in JSON)

```
"translation: {
    "athchóiriú",
    "pos": "n-masc",
    "usage": "formal"
}
```

We have seen how, in XML, every element is basically a name-value pair, where the value can be a literal value or a list of children. In JSON, every member is similarly a name-value pair. The name appears before the colon : and the value after it, where the value can be either a literal or a complex object. The underlying object model of JSON is therefore similar to that of XML. When we ignore the superficial differences in the syntax of the two languages, there are only two relevant differences in their object models (after [16] section 2.3): data elements in JSON are unordered whereas in XML they are ordered, and the keys inside a JSON object must have unique names whereas in XML the children of a parent are not required to have unique names.

In theory, it would be possible to extend the JSON language so that name-value pairs can optionally become triples consisting of a name, a value and an object containing the children. Listing 38 shows what a data fragment might look like when encoded in such an extension of JSON. This would introduce built-in support for headedness into JSON. This is, however, only a hypothetical speculation as no such JSON extension exists.

Listing 38: A hypothetical extension of JSON to support headedness

```
"translation: "athchóiriú" {
    "pos": "n-masc",
    "usage": "formal"
}
```

6.3. Headedness in YAML

A popular serialisation language which is even less verbose than JSON is YAML [17]. YAML was designed deliberately to be as human-readable and human-writable as possible. Where other languages use (curly, pointy...) brackets and quotation marks to demarcate where things begins and end, YAML uses whitespace and indentation. If data encoded in JSON look and feel like source code in JavaScript or some other C-style language, then data encoded in YAML looks and feels like source code in Python. Listing 39 shows how an entry fragment might be encoded in YAML.

Listing 39: How a translation might be encoded in YAML

```
translationContainer:
  translation: athchóiriú
  pos: n-masc
  usage: formal
```

This is undoubtedly as "unverbose" as we can get from any serialisation language. But, crucially, this still does not encode the fact that the string `athchóiriú` is the head of the whole structure. Same as in the JSON example, `translationContainer` is a purely structural element.

Like XML and JSON, YAML has no support for headedness, and the only way to encode headed structures is either to matryoshkize the data through purely structural elements, or to accept some other trade-off. The strategy of parentless sequencing and the mixed content strategy are not possible in YAML (without introducing their own purely structural markup), and the two strategies based in attributes are not possible either because there is no concept of attributes in YAML.

As a thought experiment, how would the syntax of YAML need to change to be able to accommodate headed structures? It would have to be possible for an object to have *both* a literal value *and* a list of children, like in listing 40.

Listing 40: A hypothetical extension of YAML to support headedness

```
translation: athchóiriú
  pos: n-masc
  usage: formal
```

This is illegal in YAML, but it is in fact the same syntax we have used throughout this article to illustrate headed structures. An extension like this would turn YAML into a serialization language which supports headedness.

6.4. Headedness in NVH

NVH (Name-Value Hierarchy)[4] is a less well-known markup language developed by computational lexicographers[5] in Masaryk University and in Lexical Computing, a company which makes software for lexicography. NVH is used by Lexical Computing in-house during the semi-automated production of dictionaries [4], an agile process where frequent schema updates are common.

The syntax of NVH is similar to YAML, so that an NVH document may (if certain constraints are met) also be a valid a YAML document. Additionally, NVH differs from YAML in that it implemens the proposal suggested in the previous section: an element in NVH is allowed to have *both* a literal value *and* a list of children, like in listing 40. Listing 41 shows what a complete dictionary entry looks like when encoded in NVH.

NVH is the only markup language in existence designed specifically with headedness in mind. Unlike SGML, which supports headedness at the expense of increased parsing complexity, NVH documents are as simple to parse as YAML or JSON. This is because NVH is built not on the notion of name-value pairs but on the notion of name-value-children triples.

Listing 41: An entire entry encoded in NVH

```
headword: house
  pos: noun
  phon: haus
    soundfile: house.mp3
  sense:
    definition: a built structure with walls and a roof for living in
    label: Construction
    translation: hiša
      pos: feminineNoun
    translation: dom
      pos: masculineNoun
      label: informal
    collocation: a large house
      translation: velika hiša
      example: We bought a large house.
        translation: Kupili smo veliko hišo.
```

[4] `https://www.namevaluehierarchy.org/`

[5] The author of this article is one of the creators of NVH.

7. Desiderata for a serialization language in lexicography

If we were to create an ideal markup language for lexicography, what features should the language have (in addition to support for headedness)? Which features of XML, SGML, JSON, YAML and NVH would we like to bring into this new language? This section will list some criteria and evaluate each language against them.

7.1. Avoid purely structural markup

Avoiding purely structural markup is important in lexicography for human readability and as a form of preparedness for future schema updates.

- **XML** encourages purely structural markup. To avoid it in XML, one has to resort to strategies which come with trade-offs (see section 5).
- **SGML** makes it possible to avoid purely structural markup thanks to its markup minimization features. However, this brings an increased compexity for parsing.
- In **JSON** and **YAML**, purely structural markup is practically unavoidable – although the extension proposed in sections 6.1 and 6.2, which would add headedness support to the languages, would also remove the need for most purely structural markup.
- **NVH** makes it relatively easy to avoid purely structural markup thanks to its built-in support for headedness.

7.2. Headedness

Support for headedness is obviously a high-priority requirement for a lexicographic markup language, given how prevalent headedness is dictionaries.

- **XML** has no built-in support for headedness, except when using one of the attributes-based strategies, which however comes at the expense of the ability to represent in-line markup on the either head element or on the child elements.
- **SGML** has built-in support for headedness if the encoding makes use of SGML's markup minimization features. This comes at the expense of easy machine processability: parsing SGML is a complex task.

- **JSON** and **YAML** have no built-in support for headedness either. The languages would need to be extended along the lines suggested in sections 6.1 and 6.2 in order to support headedness.

- **NVH** has built-support for headedness, but at the expense or making in-inline markup difficult: same as XML when combined with attributes-based strategies.

7.3. Explicit listing order

One requirement which is important in lexicography is preserving order. The order in which items are listed needs to fixed, remembered during parsing, and guaranteed to survive every parsing-serialisation roundtrip. Having things listed in a given order is almost always an implicit requirement when encoding lexicographic data.

- **XML**, **SGML** and **NVH** meet this requirent perfectly. The "order matters" principle is part of the design of the languages.

- In **JSON** and **YAML**, the children of a parent are not in any explicit order. For example, in JSON, every object is basically a collection of key-value pairs, and this collection is unordered. In practice JSON and YAML parsers and serializers often do preserve the order of items, but this is not guaranteed. The only way to fix the order of items is to encode them as an array (in JSON) or as a list (in YAML), which brings its own share of purely structural markup.

7.4. Non-unique child names

It is common in lexicography that a content object has multiple children of the same kind, for example an *entry* contains several *senses*, a *sense* contains several *translations*. To encode this without purely structural markup, the language has to allow the children of an element to have non-unique names.

- In **XML** and **SGML**, non-unique child names are allowed. It is possible, for example, for an `<entry>` element to have multiple children named `<sense>`, or for a `<sense>` element to have multiple children named `<translation>`.

- In **JSON** and **YAML**, non-unique child names are not allowed, and so code fragments such as listings 42 and 43 would be invalid in JSON and YAML. To remodel it into legal JSON or YAML one would need to resort to some form of purely structural markup, for example using array syntax `[...]` in JSON.

- **NVH**, in spite of its superficial similarity to YAML, does allow non-unique child names. So a code fragment like listing 43, altough invalid in YAML, is valid in NVH.

Listing 42: Invalid JSON with non-unique child names

```
{
  "sense": {
     "gloss": "completely",
     "translation": "go hiomlán",
     "translation": "go huile agus go hiomlán"
  }
}
```

Listing 43: Invalid YAML (but valid NVH) with non-unique child names

```
sense:
    gloss: completely
    translation: go hiomlán
    translation: go huile agus go hiomlán
```

7.5. Inline markup

One of XML's strong points is its good support for inline markup. Here XML shows its origins as a *markup language* (as opposed to a *serialisation language*). This heritage proved itself useful in the early stages of digitisation in lexicography when dictionary entries were treated rather like small documents, consisting of running text which needed to be marked up. *Dictionary encoding* used to be like *text encoding* in the early stages of its digitisation, and XML's support for inline markup was useful in that scenario.

Since then, dictionaries have evolved away from the text encoding paradigm. Dictionary entries have ceased to look like running text with markup and have started to look more like structured data records. Consequently, the need for inline markup has diminished. Inline markup is now used fairly rarely in lexicography. The only application where inline markup plays a role (in

some dictionaries) is to mark up the occurrences of headwords (and sometimes collocates) inside example sentences. In other wors, in-line markup is a low-priority requirement in lexicography: other requirements, such as headedness or an explicit listing order, are more important.

- **XML** and **SGML** have good built-in support for inline markup, as explained.
- In **JSON** and **YAML**, inline markup is difficult to encode as neither language has any built-in support for it. One must either matryoshkize the data (turn a string into an array of strings and objects) or invent a custom-built formalism (using some form of markdown, or stand-off markup based on start and end indexes).
- **NVH** has no built-in support for inline markup in the current form of the language, but there is a convention for representing in-line markup through stand-off annotation, as show in listing 44. The `@` character identifies the element as in-line markup of its parent, and the index number after it identifies which occurence of the substring is supposed to be marked up (in case there are multiple occurences). This convention may become part of the specification of NVH in the future.

Listing 44: NVH with inline markup

```
example: We bought a larger house in the village.
  headwordHighlight: house @1
  collocateHighlight: larger @1
      lemma: large
      pos: adj
```

7.6. Easily machine-processable

A data language is easily machine processable if it is relatively easy to write parsers for it, if the language is (in some sense of the word) *simple*. This implies a subjective judgment, but the following is probably a fair summary.

- **XML** (arguably) is easily machine-processable if one ignores optional complications such as namespaces.
- **SGML** (arguably) is not easily machine-processable due to its markup minimization features which require the parser to have access to the schema and to perform inference during parsing in order to infer closing tags and implicit elements.

- **JSON**, **YAML** and **NVH** (arguably) are easily machine-processable.

7.7. Human-friendly

We have discussed in section 2.2 how human-friendliness is important in lexicography because human editors are usually exposed to the full verbosity of the markup language. A data language is human-friendly to the extent that it is *human-readable* and *human-writeable*. What that actually means may differ from human to human, but in the author's opinion, a human-friendly language should (1) have as little syntactic punctuation (such as pointy brackets, curly brackets) as possible and (2) indicate structure by something highly visual, such as whitespace and indentation, instead of paired brackets.

- **XML** and **SGML** (arguably) possess a low degree of human-friendliness due to the fact that they contain a lot of syntactic punctuation (although some of it can be minimized in SGML) and because structure is indicated by paired tags, which may not correspond to whitespacing and indentation.

- **JSON** scores better than XML and SGML on human-friendliness because it contains less syntactic punctuation, but worse than YAML and NVH because it still *does* contain some syntactic punctuation and because structure is indicated by paired prackets.

- **YAML** and **NVH** are practically the same in this respect and both possess a high degree of human friendliness. There is almost no syntactic punctuation, and structure is indicated through indentation.

7.8. Summary: the perfect lexicographic markup language does not exist

This section has listed off a lexicographic "wishlist" of criteria for an ideal serialization language, and evaluated briefly how each language meets or does not meet the criteria. The results are summarized in table 1 for XML and in table 2 for the remaining languages.

As we have seen, there is not a single data language in existence today which would tick all the boxes on the wishlist, although NVH and SGML come close. An interesting conclusion is that XML, in spite of being widely used in lexicography, is not the best possible fit for the requirements of the field, due mainly to its lack of support for headedness.

	XML	XML[1]	XML[2]	XML[3]	XML[4]
Avoid PSM	no	yes	yes	yes	yes
Headedness	no	no	no	yes	yes
Explicit listing order	yes	yes	yes	no	yes
Non-unique child names	yes	yes	yes	no	yes
Inline markup	yes	yes	yes	no	no
Easily machine-processable	yes?	no?	no?	yes?	yes?
Human-friendly	no?	no?	no?	no?	no?

Table 1: A lexicographic scorecard for XML. "XML" means conventional XML with matryoshkization; "XML[1]" is XML with parentless sequencing; "XML[2]" is XML with mixed content; "XML[3]" is XML with children as attributes; "XML[4]" is XML with heads as attributes. Answers that assume a subjective judgment are labelled with a question mark. "PSM" stands for "purely structural markup".

	SGML	JSON	JSON[x]	YAML	YAML[x]	NVH
Avoid PSM	yes	no	yes	no	yes	yes
Headedness	yes	no	yes	no	yes	yes
Explicit listing order	yes	no	no	no	no	yes
Non-unique child names	yes	no	no	no	no	yes
Inline markup	yes	no	no	no	no	no?
Easily machine-processable	no?	yes?	yes?	yes?	yes?	yes?
Human-friendly	no?	no?	no?	yes?	yes?	yes?

Table 2: A lexicographic scorecard for languages other than XML. "JSON[X]" and "YAML[X]" means JSON and YAML with extensions suggested in sections 6.1 and 6.2. Answers that assume a subjective judgment are labelled with a question mark. "PSM" stands for "purely structural markup".

8. Conclusion

This article has challenged the age-old orthodoxy in computational lexicography that dictionary data is best encoded in XML. XML is widely used in lexicography but, on closer inspection, it turns out not to be the best fit for its requirements. We have analysed what lexicography actually needs from a markup language, with special attention to the inherent *headedness* of much of lexicographic content. We have seen how widely used languages such as XML, JSON and YAML have no built-in support for headedness, and how attemting to represent headed data in these languages results in an undesirable proliferation of matryoshkization and purely structural markup.

A schema designer who wishes to avoid purely structural markup has a number of options. Within XML, there are strategies for avoiding structural markup, but these come with trade-offs which may or may not be acceptable. Outside XML, we have shown that other well-known languages, namely JSON and YAML, are no better than XML at meeting the needs of lexicography. The conclusion is that the needs of lexicography would best be met either by a return to SGML, or by an adoption of the less well-known language NVH.

References

[1] W3C, Extensible Markup Language (XML) 1.0 (2008).
URL `https://www.w3.org/TR/2008/REC-xml-20081126/`

[2] N. Ide, A. Kilgarriff, L. Romary, A formal model of dictionary structure and content, in: Proceedings of the 9th Euralex International Congress, Stuttgart, Germany, 2000, pp. 113–126.

[3] H. E. Wiegand, Der Begriff der Mikrostruktur: Geschichte, Probleme, Perspektiven, in: Wörterbücher: Ein internationales Handbuch zur Lexikographie, de Gruyter, Berlin, 1989, pp. 409–462.

[4] M. Jakubíček, M. Měchura, V. Kovář, P. Rychlý, Practical Post-Editing Lexicography with Lexonomy and Sketch Engine, presentation at XVIII EURALEX International Congress (Jul. 2018).

[5] K. J. Carlson, The Case Against XML (2007).
URL `http://www.krisandsusanna.com/Documents/the-case-against-xml.pdf`

[6] M. Měchura, Introducing Lexonomy: an open-source dictionary writing and publishing system, Electronic lexicography in the 21st century: Proceedings of eLex 2017 conference, Leiden, 2017, pp. 662–679.
URL https://michmech.github.io/pdf/elex2017.pdf

[7] TshwaneDJe, TLex Suite: Dictionary Compilation Software.
URL https://tshwanedje.com/tshwanelex/

[8] J. Erlandsen, iLEX, a general system for traditional dictionaries on paper and adaptive electronic lexical resources, in: Proceedings of the 14th EURALEX international congress, Fryske Akademy, Leeuwarden/Ljouwert, The Netherlands, 2010, pp. 306–306.

[9] IDM, DPS User Manual.
URL https://dps.cw.idm.fr/

[10] U. Ogbuji, Considering container elements: When to use elements to wrap structures of other elements., in: Principles of XML design, IBM, 2004.
URL https://www.ibm.com/developerworks/library/x-contain/index.html

[11] ISO, Standard Generalized Markup Language (SGML) (1986).
URL https://www.iso.org/standard/16387.html

[12] C. F. Goldfarb, Y. Rubinsky, The SGML handbook, Clarendon Press and Oxford University Press, Oxford and New York, 1990.

[13] ECMA, The JSON data interchange syntax (2017).
URL http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

[14] ISO/IEC, The JSON data interchange syntax (2017).
URL https://www.iso.org/standard/71616.html

[15] N. Nurseitov, M. Paulson, R. Reynolds, C. Izurieta, Comparison of JSON and XML data interchange formats: a case study., 22nd International Conference on Computer Applications in Industry and Engineering 9 (2009) 157–162.

[16] P. Bourhis, J. L. Reutter, D. Vrgoč, JSON: Data model and query languages, Information Systems 89 (Mar. 2020). doi:10.1016/j.is.2019.101478.

[17] YAML, YAML Ain't Markup Language (YAML) version 1.2, Revision 1.2.2 (2021).
URL https://yaml.org/spec/1.2.2/